COMPOSITION OF SEMANTICALLY ENABLED GEOSPATIAL WEB SERVICES

FELIPE DE CARVALHO DINIZ
February, 2016

SUPERVISORS:

Dr. ir. R. A. de By

Dr. J. M. Morales
Dr. ir. R. L. G. Lemmens

COMPOSITION OF SEMANTICALLY ENABLED GEOSPATIAL WEB SERVICES

FELIPE DE CARVALHO DINIZ
Enschede, The Netherlands, February, 2016

Thesis submitted to the Faculty of Geo-information Science and Earth Observation of the University of Twente in partial fulfilment of the requirements for the degree of Master of Science in Geo-information Science and Earth Observation.
Specialization: Geoinformatics

SUPERVISORS:

Dr. ir. R. A. de By

Dr. J. M. Morales
Dr. ir. R. L. G. Lemmens

THESIS ASSESSMENT BOARD:

Prof. Dr. M.J. Kraak (chair)

Prof. Dr. A. Wytzisk (Bochum University of Applied Sciences)

ABSTRACT

Applications such as disaster and emergency management require near-instant access to data from different sources to make decisions and take actions rapidly. Although previous experiences have shown that the geospatial data availability in many scenarios is not a problem, this data is usually provided from multiple heterogeneous data sources distributed over the web, requiring the data to be discovered and integrated. Also, the data not necessarily is ready to be used, requiring preprocessing steps to turn it into actionable information.

Although OGC standards have made significant progress towards syntactic interoperability of services and feature-level data access, they do not solve semantic heterogeneity problems. At the same time, Semantic Web Technology can provide semantic interoperability but has had a slow uptake, since it is complex and does not follow current trends for data formats and access, thereby heading in an opposite direction of what developers and users expect. Even for services that provide semantically enabled data, service discovery and manual composition in a distributed environment are still time-consuming and error-prone. Typically, one must select multiple data provision services, and apply the processes with the need to understand their functionality and input/output restrictions. As a consequence of these difficulties, there is a lack of tools and methods to facilitate the construction, verification and execution of composition of geospatial web services.

This research project aims to address limitations on service composition verification, execution, and sharing. We develop a theoretically founded environment in which geospatial service composition can be verified. The theory formally defines what composability of semantically enabled geospatial web services is, providing a base for the development of an algorithm for verifying the composability of services, which grants compile-time protection against a class of errors. This protection allows not to run an invalid composition that could lead to unnecessary long running time as well high usage of the server processing capabilities. This thesis also provides guidelines for defining lightweight services for sharing, processing geospatial data and for cataloguing geospatial data and services based on OGC Standards, and on current trends in web technology, such as REST architecture, WebSockets, and JSON-LD. We expect that the proposed services will facilitate the implementation of OGC standards and lower the need for third-party software. We also extend the OGC services to be semantically enabled, providing service functionality descriptions by using the Hydra Core Vocabulary. These descriptions allow applications to discover and use web services without being specifically programmed for each service, lowering human interaction, and allowing the implementation of a generic client. Lastly, we develop JSON-W, a notation to describe and share compositions which allow a lossless roundtrip to RDF serialization. This makes the compositions interoperable and at the same time maintain the ease to use and transfer to the Web of the JSON format.

Keywords

Web service composition, Orchestration, Geoprocessing workflow, Web Processing Service, RESTful API, Semantic description

ACKNOWLEDGEMENTS

I would like to thank the Brazilian Army and the Board of Geographic Service (DSG), who had faith in me, giving me the opportunity of pursuing a Master of Science degree.

Also, I want to express my sincere gratitude to my first supervisor Dr. Ir. Rolf de By for his guidance, pieces of advice, and for the time spent reviewing endless drafts. The discussions and constructive criticisms throughout this thesis made me a better engineer and researcher.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE OF ABBREVIATIONS

API	Application Program Interface
BBOX	Bounding Box
BPEL	Business Process Execution Language
CORS	Cross-Origin Resource Sharing
CRS	Coordinate Reference System
CSW	Catalogue Service for the Web
EWKB	Extended Well-Known Binary
EWKT	Extended Well-Known Text
ExtGeoJSON	Extended GeoJSON
GML	Geography Markup Language
GPX	GPS Exchange Format
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
ISO	International Organization for Standardization
JSFE	JSON Serialization for Filter Encoding
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
JSON-P	JSON with Padding
JSON-W	JavaScript Object Notation for Workflows
KVP	Key-Value Pair
OC	Orchestration Client
ODATA	Open Data Protocol
OE	Orchestration Engine
OGC	Open Geospatial Consortium
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
OWS	OGC Web Services
QUDT	Quantities, Units, Dimensions and Types Ontologies
RDF	Resource Description Framework
REST	Representational State Transfer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SOS	Sensor Observation Service
SPARQL	SPARQL Protocol and RDF Query Language
SWRL	Semantic Web Rule Language
TRS	Temporal Reference System
TWKB	Tiny Well-known Binary
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
UUID	Universally Unique Identifier
W3C	World Wide Web Consortium
WCS	Web Coverage Service
WFS	Web Feature Service

WKB Well-known Binary
WKT Well-known Text
WPS Web Processing Service
WSDL Web Service Definition Language
XML Extensible Markup Language

Chapter 1
Introduction

1.1 BACKGROUND AND MOTIVATION

Applications such as disaster and emergency management require near-instant access to data from different sources to make decisions and take actions rapidly. Geospatial data such as road network, land use, demographic data, and satellite images play a significant role to improve situational awareness for decision makers. Although previous experiences have shown that the geospatial data availability in many scenarios is not a problem [1], this data is usually provided from multiple heterogeneous data sources distributed over the web, requiring the data to be discovered and integrated. Also, the data not necessarily is ready to be used, requiring pre-processing steps to turn it into actionable information. Several studies report that data integration is one of the most immediate and limiting challenges [2, 3], stating that data discovery and integration in heterogeneous environments can take up to 50% of user time [4, 5]. The user needs to put a considerable effort in integrating spatial data formats and schemas, based on concepts typically from different fields of study and natural languages.

The Open Geospatial Consortium (OGC) Standards provide service definition guidelines, solving problems of syntactic and structural heterogeneity among different data sources by providing services with homogeneous access points and data formats [6]. For the scope of this thesis, three types of web service are considered: Web Feature Service (WFS), Web Processing Service (WPS), and Catalogue Service for the Web (CSW). The WFS standard provides feature-level data access, saving the users from downloading the entire dataset and then selecting the subset that is required for analysis. The data is usually encoded in the Geographic Markup Language (GML), an Extensible Markup Language (XML) grammar capable of expressing geospatial features. The WPS standard provides means of wrapping any computational process, specifying the interface for input and output, and indicating how users can execute the process [7]. The CSW standard defines the implementation rules of a catalogue of geospatial records in XML, which provides the means of discovery metadata about geospatial data and services [8].

Although OGC standards have made significant progress towards syntactic interoperability of services and feature-level data access, they do not solve semantic heterogeneity problems [9]. The standards do not provide machine-interpretable descriptions, which in turn does do not allow applications to decide autonomously in which actions to participate. This lack of semantics in the data and service descriptions is considered a primary limitation for achieving semantic interoperability, i.e., the exchange of information in an unambiguous way, with the appropriate understanding of the meaning of the data [10]. Semantic interoperability allows the use of the data in contexts that it was not originally created and also be used by autonomous clients, that, for example, can reason over multiple service descriptions and assist a user in finding which service best fits its requirements.

To achieve semantic interoperability, alternatives in Semantic Web Technology have been proposed. The Semantic Web is an extension of the Web based on World Wide Web Consortium (W3C) standards that facilitate data sharing and reuse across applications by providing accurate and

unambiguous definitions with data schemas. One way of achieving the vision of the Semantic Web is Linked Data, a group of best practices to publish and interlink structured machine-readable data on the web [11]. In summary, best practices include standard identification of resources (URI); use of the Resource Description Framework (RDF) graph-based data model to structure and interlink data; and use of the SPARQL Protocol and RDF Query Language (SPARQL), the standard query language for RDF. Linked Data gives a well-defined machine-readable meaning to resources, which allows the representation and inference of relationships in the data, the resolution of ambiguities, and enabling of data interoperability at a semantic level [12].

In contrast to what one would expect, interest in Semantic Web technology appears waning [13] and is known of only a small number of enterprise applications that make use of RDF/XML serialization, the encoding of RDF data in XML [14, 15, 16]. Several factors can be attributed to this low uptake, such as SPARQL syntax being complex and requiring specialized knowledge, SPARQL endpoints being costly in computational cost at the server side and create server availability issues when facing multiple concurrent users. Although the Semantic Web was conceptualized for querying data from multiple sources, federated query processing is still slow, and this state of affairs is aggravated by the low availability of endpoints. In 2012, OGC standardized GeoSPARQL, an extension of SPARQL for querying spatial data encoded in RDF [17], which encouraged sharing spatial data in this format. However, triple stores, the specialized storage form for RDF data, are not so well-developed compared to relational databases to handle spatial data.

One factor that explains the low uptake of OGC Standards, and also applicable to the Semantic Web Technology, is the fact that they do not follow the current trends of technology for data formats and data access. Both heavily use XML technology for data encoding and descriptions, ignoring formats such as JSON, and for data access, they are mostly based on Remote Procedure Call (RPC) and the Simple Object Access Protocol (SOAP) [6]. Figure 1.1 presents the use of search terms SOAP compared to Representational State Transfer (REST), a more modern approach for data access, in the last ten years [18].

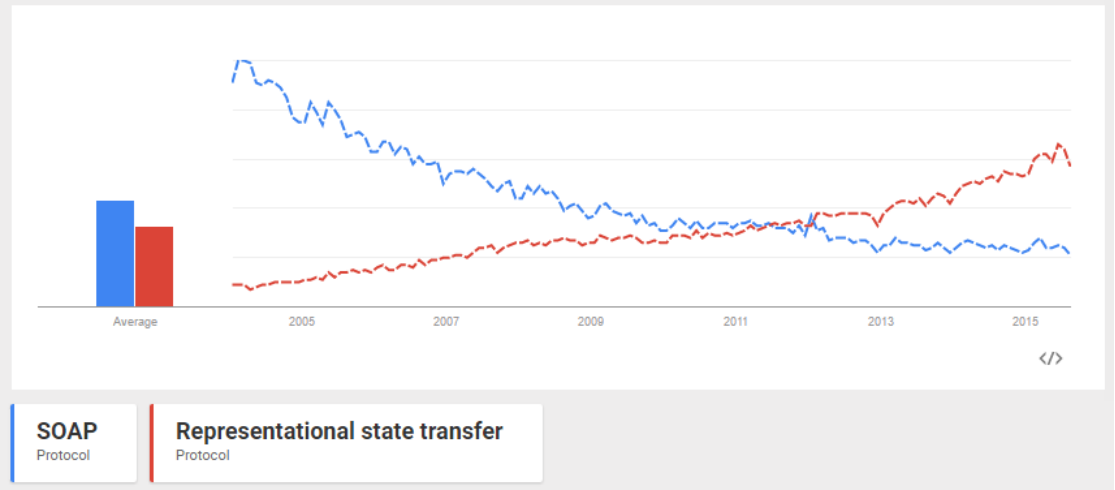


Figure 1.1: Trend of SOAP and REST in Google searches

Even for services that provide semantically enabled data, service discovery and manual integration of sources in a distributed environment is still time-consuming and error-prone. Typically, a data provision service must be manually selected, the required data must be downloaded, and the process must be applied, with the need to understand their functionality and input/output restrictions. These tasks can be minimized by composing web services. Service composition refers to the process of selecting and assembling a meaningful combination of services to solve a specified

problem. Different services are pipelined in such a way that the output of one service can serve as input to another service. Ideally, this composition should be assembled and verified automatically, however, in present state-of-the-art, this procedure is done mostly with intensive human interaction.

To address the limitations on service composition, this research proposes modifications to the definition of OGC services for sharing, cataloguing and processing geospatial data. The goal is to enhance the capabilities of verifying, executing, and sharing the composition of those services. The modified services are defined on the basis of contemporary web technology, aligned with current standards, and are capable of providing machine-readable descriptions of their functionality. The vision is that applications should autonomously be able to discover, use and compose multiple services without the need for creating a specialized routine for each service. The here proposed service type is semantically enabled, meaning that it provides data with explicit semantics for the feature types, feature instances, and service functionality. The proposed services have five advantages compared to current implementations:

1. By using contemporary and accessible web technology, we facilitate the implementation of OGC standards and lower the need for third-party software.
2. By using semantically enabled descriptions, the functionality of the service can be discovered and invoked without human interaction.
3. Service composability is supported by a formal theory, providing means of unambiguously defining and verifying the validity of service compositions, and possibly, over time, leading to more robust service design practices.
4. Composition execution is based on the WPS asynchronous execution model, which allows workflow bandwidth usage optimization by using the mechanism of data transfer by reference.
5. By using the mechanism of metadata propagation and the proposed JSON Workflow representation the compositions can be shared in an interoperable manner.

As proof of concept, an application that allows geospatial service composition based on the proposed services was implemented. The application let the user discover available services and assemble them in the desired combination, share the composition in an interoperable manner, and orchestrate the service execution in an asynchronous way. The system assists the user by providing compile-time protection against errors originating from invalid composition between services.

1.2 RESEARCH OBJECTIVES

The main objective of this research project is to develop a theoretically founded environment in which geospatial service composition can be verified, executed, and shared. The services aim to provide geospatial data sharing, cataloguing and processing capabilities in support of applications that have rapid decision-making processes. The theory formally defines what composability of semantically enabled geospatial web services is, providing a base for the development of an algorithm for verifying the composability of services, which grants compile-time protection against errors.

The main objective can be divided into the following sub-objectives:

1. Propose lightweight, syntactically interoperable services for sharing, cataloguing and processing geospatial data using contemporary web technology, and align these with current OGC and ISO standards.
2. Propose a theoretical foundation for verification of composability of geospatial web services.

3. Define how the proposed services can be made self-describable with machine-readable, semantically enabled annotations.

4. Define mechanisms for describing, sharing and executing geospatial service compositions.

Questions related with sub-objective 1:

1. How can current standards be modified to align with contemporary web technology?

Questions related with sub-objective 2:

2. Which forms of composition can be recognized, and for each of these, to what extent can we decide on their validity?

3. How to verify the validity of the composition of services?

Questions related with sub-objective 3:

4. How can OGC services be extended to be semantically interoperable?

5. How can geospatial web services functionality be described in a machine-readable and machine-interpretable way?

Questions related with sub-objective 4:

6. How can the metadata of individual services be combined and propagated to generate descriptions for a composite service?

7. How can geospatial service composition be described and shared in an interoperable way?

8. How can bandwidth usage in the execution of a composition be minimized?

1.3 RESEARCH APPROACH

This project has been divided into three activities: proposal of lightweight semantically enabled geospatial web services, the definition of a theory for verifying the composability of web services, and definition of mechanisms to describe, share and execute composition of geospatial web services.

The first activity deals mainly with requirements analysis for web services based on standards for data encoding, sharing, processing and cataloguing. The current trend of using web technology is also considered, in an attempt to avoid outdated solutions. The outcome of this activity is the identification of the required functionality for WFS, WPS, and CSW, and a proposal for changes in the way in which the services are semantically enabled, in the sense that they are capable of providing semantically enabled spatiotemporal data and descriptions. The new functionality is added to the OGC services, so they follow the REST architecture, use WebSockets, and use JSON-LD as the main format for sharing data and metadata.

The second activity focuses on the creation a formal theory for composability of services. The goal of this theory is to define what composability of semantically enabled geospatial web services is, describe how the validity of composition is verified and how the result of the composition of services can be inferred. The composability verification is divided into levels of composability based on the expressiveness of the verification mechanism:

1. Structural Composability, which abstracts the service composition as a graph, where the nodes are services and edges connections between inputs and outputs, and evaluates the general interactions between the inputs and outputs of services without taking into account specifics of each them. The goal to verify invalid interactions such as cycles, duplicates, verify the existence of all required connections, and validity of loops.

2. Static Syntactic Composability, in which Geospatial web services and their service composition are seen as a domain of typed functions, in a way that the output of a component service can only become the input of the next service if their data types are identical or the output data can be automatically coerced to the data type of the input parameter.
3. Dynamic Syntactic Composability, which refers to verify restrictions in the data being transferred that are not covered by Static Syntactic Composition, such as interactions between inputs, coordinate systems, and time zones. This verification is done based on precondition checking of Hoare logic [19].
4. Semantic Composability addresses the meaning of the data and computational processes, verifying whether the combination of processes can derive a meaningful result, or it produces the intended result of the user.
5. Qualitative Composability, which evaluates the composition against user requirements related to non-functional characteristics, such as response time, service availability, cost, security, legal rights, and quality of operations.

The last activity addresses the definition of mechanisms for describing, sharing and executing service composition. For descriptions of service compositions, we define a mechanism for deriving metadata from a composition based on the description of individual services. This mechanism allows opaque compositions that are indistinguishable from WPS implementations. For sharing, we develop a workflow notation inspired by the Business Process and Model Notation (BPMN) specification [20]. It uses concepts of Semantic Web Technology such as RDF and ontologies to ensure the workflows can be shared in an interoperable manner, and JSON-LD framing to ensure the simplicity of use and sharing in a Web environment without losing the expressiveness of RDF. Regarding execution, we propose and implement a mechanism for executing the composition of the asynchronous execution model of the WPS standard. This mechanism leverages the modifications proposed in the WPS standard that an asynchronous execution returns a result that is indistinguishable from a WFS resource, harmonizing the passing of data between OGC web services. The proposed execution mechanism allows bandwidth usage optimization by using mechanisms of data transfer by reference, instead of by value.

1.4 RESEARCH RELEVANCE

OGC standards such as WFS and WPS are considered complex and developers need to rely on third-party software implementation. The standards are based on XML encodings and the RPC/-SOAP protocols, a web technology that has largely been replaced in the last few years. This thesis provides guidelines for defining lightweight services for sharing, processing geospatial data and for cataloguing geospatial data and services based on OGC Standards, and on current trends in web technology, such as REST architecture, WebSockets and JSON as main data transfer format. In this way, the potential adoption of the proposed services is maximized. Also, the OGC Standards aim to solve syntactic interoperability, not semantic. This thesis provides ways to extend OGC services to provide semantically enabled data and also to provide semantically enabled descriptions of the service functionality. These descriptions allow applications to discover and use web services without being specifically programmed for each service, lowering human interaction.

For Geospatial Web Service Composition several advancements are made in the fields of workflow verification, metadata generation, metadata sharing and execution. For workflow verification, four theoretically founded levels of composability are developed and discussed. The focus of the theory of composability is compile-time verification, which prevents to run an invalid composition that could lead to unnecessary long running time as well high usage of the server processing

capabilities. The composability theory covers metadata propagation, which allows the generation of the specification of composite services. For sharing, we develop a JSON-LD workflow representation capable of describing the interaction between web services, as well as that it covers the use cases of subgraph, conditional and loop, and capable of being stored and shared using the CSW specification. The developed notation for describing and sharing workflows allows a lossless roundtrip to RDF serialization, making the workflows interoperable and at the same time maintaining the ease to use and transfer to the Web of the JSON format. To assist the use of the JSON Workflow notation, a graphical interface is implemented that allows the user to build the composition visually. For execution, we develop an orchestration engine, implemented as a WPS, and capable of executing WPS processes in the asynchronous model, which reduces bandwidth usage by not requiring the orchestration service to handle the data between services.

1.5 THESIS OUTLINE

This thesis adopts the following structure:

Chapter 1 provides a general introduction to this thesis through a background and motivation and stating the research objectives and research approach.

Chapter 2 discusses how OGC web services can be modified to be based on contemporary web technology such as REST, WebSockets, JSON and aligned to current standards. The following services are considered: WFS, WPS, and CSW.

Chapter 3 discusses the possible forms of service composition that can be recognized and introduces theoretically founded methods to verify its validity. Also, the first level of composability is discussed, Structural Composability.

Chapter 4 discusses the next level of composability, Static Syntactic Composability, which abstracts geospatial web services and service composition as a domain of typed functions, providing a mechanism for type checking and type propagation.

Chapter 5 discusses Dynamic Syntactic Composability and Semantic Composability.

Chapter 6 provides the definition of the metadata that allows applications to use web services without being specifically being programmed for each service, which allows the creation of generic clients. Also, services are extended to serve JSON-LD for data and metadata, turning OGC Services into semantically interoperable services. It also introduces the mechanism of metadata propagation, used to generate dynamically metadata for WPS results. Moreover, based on the previous concepts, it introduces a Workflow notation that allows the composition to be described and shared in an interoperable matter.

Chapter 7 discusses the implementation of the proposed services and a generic client which is capable of processing those services metadata allowing seamless interaction with different OGC services. Also is discussed the implementation of a graphical user interface to assist users to build, verify, and execute service compositions.

Chapter 8 gives a summary of the thesis, answering the research questions, also reflecting on the limitations and providing recommendations for future work.

Chapter 2
RESTful OGC Services

2.1 INTRODUCTION

The OGC Reference Model [21] states that the standards should reflect the best engineering practices, however, the OGC services are still heavily based on RPC/SOAP/XML, the most commonly used technology in the year that the first versions of the standards were developed. Currently, OGC standards are heading in the opposite direction of what developers and users expect, not yet incorporating de-facto standard technology such as JSON, WebSockets, and REST, which could extend the use cases and adoption of the services.

The complexity of the OGC standards forces developers to rely on third-party implementations. Even when the application requires simple functionality, any full implementation of the standard will lead to a heavy and complex service. Also, as geospatial information is used in many domains, it is impossible to predict and prepare for every possible user requirement. This creates the need for flexible and extensible services, in which developers can choose the desired functionality and extend it to fulfil OGC requirements. This conceptualized service can be implemented in an incremental way, and be documented with flexible metadata that can express the functionality of the service with the required granularity by the developer. Also, the mechanism for passing data between OGC web services such as WFS and WPS is not harmonized, causing difficulties in the composition of such services. In this chapter, we propose a modification of the WPS standard so that asynchronous service execution returns a result that is indistinguishable from a WFS layer, which allows composition bandwidth usage optimization by using mechanisms of data transfer by reference, instead of by value.

In this chapter, is discussed how the OGC services can be modified to be scalable, flexible, and lightweight, following current web technology. Section 2.2 introduces the architecture currently used in OGC services, and compares it with current web technology trends. Section 2.3 gives a brief description of the REST architectural style, while in Section 2.4 we propose RESTFul bindings for the Web Feature Service, Web Processing Service and Catalogue Service of the Web. Section 2.5 discusses the use of WebSockets in OGC standards, complementing use cases in which the application of REST architecture leads to inefficient implementation. Lastly, Section 2.6 addresses the use of JSON in OGC standards, both for data encoding and metadata.

2.2 OGC ARCHITECTURE

The OGC services follow a Service-Oriented Architecture (SOA), meaning that they are loosely coupled, self-contained, have a uniform means to interact with and discover its capabilities. The OGC specification standardizes the interaction with the web services by usually providing three different binding styles: HTTP GET, HTTP POST, and SOAP, where SOAP also uses the HTTP POST as a communication protocol. The requests are made in a Remote Procedure Call (RPC) style, where the service has one entry point URL, and different interactions are realized with parameters provided with the base URL. RPC uses the HTTP protocol for communication, how-

ever, does not respect the semantics defined in the standard. The OGC specification also standardizes data formats, being XML, and one of its grammars, GML; together these are the most used formats for both metadata and data distribution. By providing homogeneous access points and data formats, OGC services are syntactically interoperable.

2.2.1 Current Web Technology Trends

The general trend of web technology is to simplify both utilization and development of applications to ensure wider adoption. By using outdated technology, the number of users and developers is effectively limited, and this lowers the applicability of the standard. The first factor that we consider is the trend in data formats. OGC standards are mostly based on XML, a format that presents a constant decline in usage in recent years. Figure 2.1 illustrates the trends of use of the search term "XML Application Program Interface" (API) compared to "JSON API" in the last decade [18]. Obviously, there is increasing use of JSON, and the continuously decreasing use of XML. Figure 2.2 illustrates the relative frequency of use of data formats in Internet APIs. The information was extracted from the website ProgrammableWeb [22], and it refers to the distribution in January 2016.

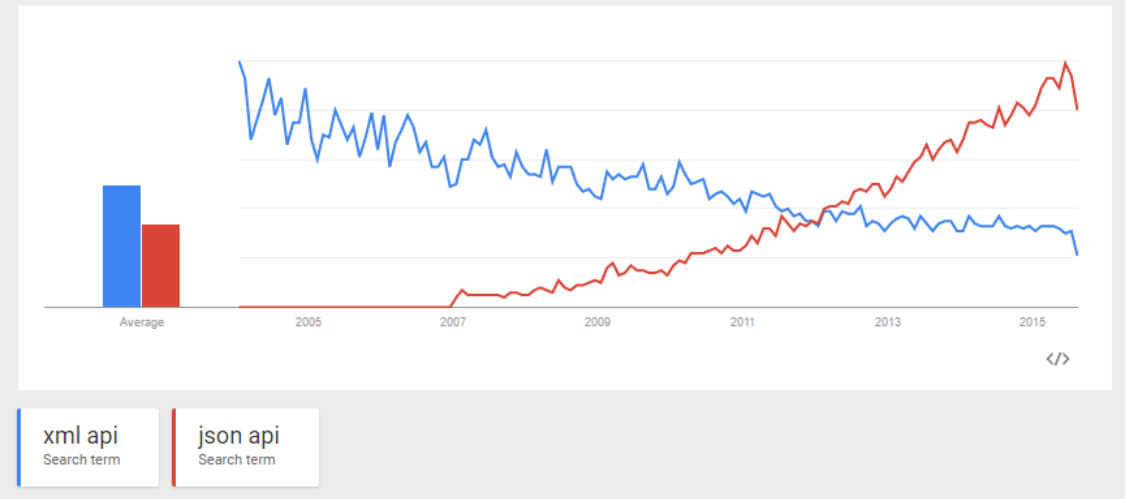


Figure 2.1: Trends of XML and JSON API in Google searches

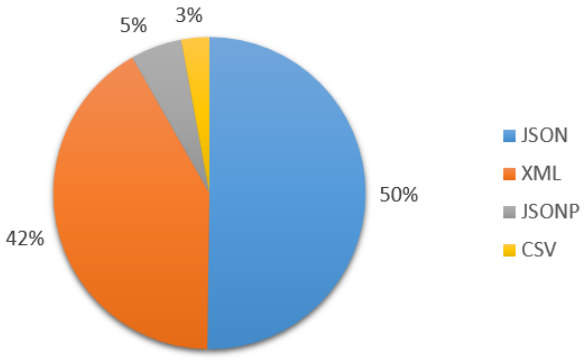


Figure 2.2: Internet API relative frequency by data format

It is not only important to look at trends in data formats, but also at those in data access. The OGC Standards interfaces are implemented through Remote Procedure Call (RPC) and the Simple Object Access Protocol (SOAP) [6]. Recently, several companies, including Google, Facebook and Yahoo, have deprecated the use of their SOAP-based interfaces, and migrated to Representational State Transfer (REST) architecture, which facilitates both the use and the development of applications [23]. Figure 2.3 presents the relative frequency of the use of Internet APIs by technology. The information was extracted from the website ProgrammableWeb [22], and it refers to the distribution in January 2016.

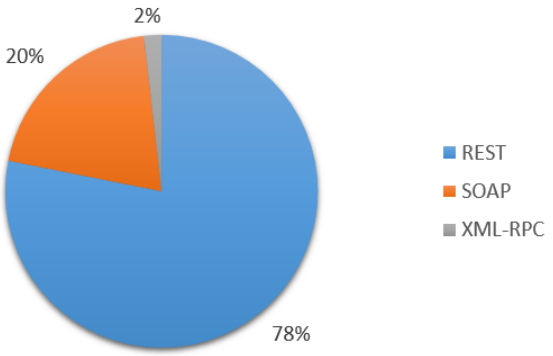


Figure 2.3: Internet API relative frequency by technology

This change can be attributed to the ease of implementation and use of REST, as it is intended to be lightweight, and it does not cover all functionality of SOAP. The learning curve to work with REST is less steep, as, for less complex applications, it requires a shorter chain of tools compared to SOAP.

Other advantages of REST architecture are performance; allowance to use the HTTP Cache; flexibility, by the use of hypermedia controls the URLs can change without breaking clients; and the possibility of working with any data format, as opposed to SOAP, which is restricted to XML. This is desirable as JSON is the primary data format used for asynchronous communication between server and browser. JSON is easier to parse in a browser, and XML is of greater complexity and typically has a larger payload. Section 2.3 addresses more in-depth the REST architectural style, and in Section 2.6 we address the use of JSON in OGC standards.

2.2.2 Previous Approaches

Based on the above-discussed limitations it appears desirable that a profile of the OGC Standards be developed that addresses the use of JSON as an exchange format and the creation of RESTful bindings. Currently, there is momentum within the OGC community to address this issue. For instance, in Testbed 11 [24] and Testbed 10 [25] discussions are ongoing on how JSON and GeoJSON can be adopted across different OGC services. In 2011, OGC formed the RESTful Service Policy Standard Working Group for the purpose of deriving requirements and recommendations for RESTFul OGC Services [26], lately adopting REST as one part of the Web Map Tiling Service (WMTS) specification [27]. Also, three best practices documents were released involving REST bindings: the RESTful Encoding of OGC Sensor Planning Service for Earth Observation Satellite Tasking [28]; RESTful Encoding of Ordering Services Framework for Earth Observation Products [29]; and OGC Download Service for Earth Observation Products [30]. Currently, OGC is developing the OGC SensorThings API standard [31] that will also be based on the REST architecture. Also worth to mention is that the WPS specification states that a REST-oriented interface

should be considered in case OGC specifications progress towards REST [7].

Another attempt outside OGC that can be cited is the proposal by Esri to adopt the GeoServices REST Specification [32] as an OGC standard, however, the request was eventually withdrawn [33]. Also, there is a submission to OGC of change requests to add REST bindings for WFS 2.0, however, this request is pending [34]. The proposal has several drawbacks that are addressed in this thesis. We discuss these in the next paragraph.

The following drawbacks can be identified. *Standardization of URL syntax breaks the flexibility of the REST architecture*: in this thesis, we propose to include at server-side a series of hypermedia descriptions that the client can discover at run-time, so as to understand the possible interactions with the service. The *use of HTTP PUT* causes the need to pass the entire resource representation in an update request: we propose to use HTTP PATCH instead. The use of OPTIONS to discover available methods and representations is not recommended, since OPTIONS cannot be cached by the client in the standard HTTP caching specification. Lastly, and typically, *transactions are specified in a non-RESTFul way*: we propose in this thesis a compliant specification based on ODATA Batch Processing.

In Testbed 11 [35], another RESTful architecture is presented focusing mostly on the upcoming WFS 2.5 standard. In this document URLs are not treated as opaque, providing best practices of how to design URL paths and defining URL templates. In this thesis URLs are seen as opaque, not having any special meaning associated, the documentation is responsible for expressing what are the contents of the resource, not needing any standardization in the URLs. Also, the document is limited in three aspects: the canonical representation of features is still GML 3.2, not being flexible to accept JSON; the discussion is limited only to simple queries, not discussing how to deal with complex queries; and it just discusses simple transactions, not discussing how to do multiple transactions in an atomic way. In this thesis, all these subjects were discussed.

We identified six research papers relevant to this discussion. Granell et al. [36] compare the OGC Web Processing Services with REST constraints, however, the authors do not propose RESTful bindings. Foerster et al. [37] proposed RESTful bindings for WPS, however, they do not provide an explanation for all WPS interactions, they also standardize the possible URLs, breaking the flexibility of REST, and they provide redundant URLs for interaction with inputs and outputs. Jiang et al. [38] propose RESTful bindings for CSW, which uses a middleware to convert the legacy catalogue to a RESTful architecture. This middleware requires that all URLs be standardized and creates an overhead in the communication to and from the service, also was not discussed how complex queries that involve Filter Encoding can be handled in the REST architecture. Mazzetti et al. [39] propose REST bindings for WCS, and Janowicz et al. [40], and Page et al. [41] propose REST bindings for SOS. However, we will not consider WCS and SOS in this thesis. Also, there are available implementations of REST bindings for OGC services, such as 52°North SOS RESTFul Extension [42] and the Sensor Web REST API [43].

2.3 REST ARCHITECTURAL STYLE

Representational State Transfer (REST) is an architectural style described by a series of constraints applied to interactions between components and data elements [44]. It uses the Hypertext Transfer Protocol (HTTP) [45] as an application protocol, which defines the semantics of the actions available to manipulate the resources. The goals of the REST architecture are performance, scalability, simplicity, modifiability, visibility, portability, reliability of the web service.

REST uses a Resource-Oriented Architecture (ROA), which works in a different granularity than Service-Oriented Architecture (SOA). In ROA, the concern is request addressing for resource instances, and in SOA, it is the creation of request payloads for service instances. Also, in SOA,

there is one endpoint address per service, in comparison to one address per resource in ROA.

The REST constraints defined in Fielding, 2000 [44] are summarized below. A service that fulfils all constraints is considered *RESTful*.

- Client-server: The idea is a separation of concerns between the clients and the server, where the server handles data storage and processing, and the client handles the user interface functionality, increasing the portability. This allows the server to evolve independently from the client as long as their interface is not altered.

- Stateless: The client-server communication is restricted in such a way that no session state is stored on the server side between requests. This forces all the necessary information to a request be stored in its contents, and the client to hold the session state. Since servers are not concerned with the client state, the server implementation is simpler and more easily scalable.

- Cacheable: The response to a request must be implicitly or explicitly defined as cacheable or non-cacheable. This allows clients to reuse static information, partially, or completely eliminating some interactions between the client and the server, which further improves scalability and performance. HTTP provides a cache mechanism by means of standard headers.

- Layered system: Servers and clients cannot distinguish whether there are intermediary layers between themselves.

- Code on demand (optional): Client functionality may be extended by transferring code to be executed on the client side.

- Uniform interface:

  - Identification of Resources: Individual resources are uniquely identifiable by a URI.
  - Manipulation through representation: Exists a conceptual separation between resources of the service and the representation of the data returned to the client. The same resource can have multiple representations such as JSON, XML, and others. In the HTTP standard, the mechanism to serve the resource in different representations is called *Content Negotiation*.
  - Self-descriptive messages: The message data and metadata should include enough information for the resource to be understood syntactically and semantically.
  - Hypermedia as the engine of application state (HATEOAS): Except for the entry point of the service, the client does not assume any action/resource available in the server. All the available operations are discovered at run-time.

2.4 RESTFUL OGC SERVICES

This section discusses the possibility of RESTFul bindings for OGC Services, focussing on the WFS, WPS, and CSW standards. The reader should bear in mind that no URLs for interactions with the services are specified in this thesis. Instead, they can be discovered by the client at run-time using resource representations (hypermedia links) returned by the service. This approach is known as the HATEOAS principle, which will be described in Chapter 6.

2.4.1 Web Feature Service

The Web Feature Service (WFS) offers methods to retrieve, create and update geospatial data independently of the underlying data source [6]. This service also contains elements of a projection service and a format conversion service. The WFS service complies with some of the constraints defined in the REST Architectural Style. WFS, and, in fact, all OGC Services, are Client-Server and Stateless.

The WFS service basically behaves like a mediator between the client and the underlying data source, in this way we can consider that it follows a Layered Approach. The Cache and Code-on-demand constraints are not followed. The lack of caching in all bindings of OGC Services greatly reduces the performance and scalability of the service. Regarding Code-on-demand, it is an optional constraint in REST, and it is not applicable in the WFS context.

One of the most important constraints on the REST architecture is the Uniform Interface. As OGC is based on RPC and SOAP, it does not comply with this constraint. All the requests are tunnelled from a unique URL instead of having separate URLs for each resource, which in turn reduces caching possibilities. Additionally, no hypermedia controls are provided, and there is no notion of manipulation through representation. The WFS specification provides self-descriptive messages. However, they are limited and mostly rely on standardized URL parameters.

The WFS uses a Service-Oriented Architecture while REST services follow the Resource-Oriented Architecture. In this way, it is not appropriate to just implement REST bindings: the WFS specification is required to be seen through the lens of ROA. The WFS specification defines three conformance classes regarding the binding styles: HTTP GET, HTTP POST, and SOAP. All requests are made in RPC style, having a central URL where all requests are made, and parameters used to distinguish the actions being performed. Request parameters can be passed using two types of encodings, Key Value Pair (KVP) and XML. The KVP encoding is used in conjunction with HTTP GET bindings while XML is used with HTTP POST and SOAP. In the case of SOAP, the POST method is also used, the only difference is that the payload of the request is a SOAP XML.

In the HTTP specification, GET is defined as a safe and idempotent method [45], meaning that it guarantees no state change on the server and several requests have the same effect as a single one. In the WFS specification, this is not followed, as one can use a GET request to create, delete and update features. Also, in the HTTP specification, POST is neither safe nor idempotent method, meaning that it cannot be used for retrieving features. Also, the semantics of each operation should be considered; the GET operation has the semantics of retrieving a resource while POST has the semantic of creating a resource. This is not followed by the WFS specification. To fulfil a REST architectural style, the WFS specification should correctly follow the HTTP specification, which currently is used only as a transport protocol. The RESTful bindings for WFS is based on 4 HTTP methods:

- GET, used to retrieve features from one layer, and to retrieve metadata.
- POST, used to insert one or more features into one layer.
- DELETE, used to remove a single feature of one layer.
- PATCH, used to update a single feature from one layer.

The method PATCH is used instead of PUT due to its extended semantics, where PUT can only be used to update an entire representation while PATCH can be used to update the representation partially. The developer can choose based on his needs which method best fits his application.

For transactions that require interaction with more than one resource, one uses Batch Processing as specified in the ODATA protocol.

The Open Data Protocol (ODATA) is an OASIS standard defining best practices for consuming and building REST services [46]. In ODATA, Batch Processing is defined, in which multiple operations can be performed on a single HTTP request by encoding the series of operations as a Multipart MIME message [47], and submitting it to the server with a POST operation. The ODATA Batch Processing is particularly interesting due to its capabilities of providing atomic transactions [48]. The idea of applying ODATA Batch Processing in OGC Standards to handle multiple operations over a single HTTP request was first introduced in the SensorThings API Standard [31].

Aside from following the HTTP specification, the services should also adhere to the notion of resources. In the WFS, all requests are tunnelled from a single URL, and parameters are used to differentiate between actions and feature layers that are being manipulated. In ROA, each resource requires having a specific URL, with which the client can interact. Also, there is a notion of representation of a resource, so that the same resource can be retrieved in different formats through the same URL. In the WFS, each available feature layer is a resource. Table 2.1 exemplifies a comparison between the KVP bindings in WFS specification, and a request in REST architecture. In this case, *lakes* is a resource that has a specific URL in the REST bindings, and content negotiation is used to retrieve the representation of the resource in GeoJSON format.

Table 2.1: Example of resource URL for WFS

OGC KVP	http://www.example.com/wfs.cgi?SERVICE=WFS& VERSION=2.0.0& REQUEST= GetFeature& TYPENAME=Lakes& OUTPUTFORMAT=GEOJSON
REST	**GET** http://www.example.com/wfs/2.0.0/lakes/ **Accept:** application/vnd.geo+json

Possible methods and parameters can be defined individually for each layer, providing the developer with a flexible granularity for the application according to his needs. Also, to enhance flexibility, URLs should not be standardized in any way, the service metadata should inform the client about server capabilities. This enhancement in the self-descriptive messages based on hypermedia controls is presented in Chapter 6.

The important disadvantage of the proposed RESTful bindings is that, as REST is a simple architecture compared to RPC/SOAP, it cannot handle all the use cases described in the standard, especially concerning complex queries within one resource and queries that involve more than one resource. For the cases of complex queries within one resource, the service can be extended with an additional resource for Stored Queries. The client will then make a POST request to this resource with the payload containing the query in a standard language such as Filter Encoding (or the JSON serialization presented in Section 2.6.2), and the server will return the Location Header with a URL where the client can access the result of the query. The drawback of this approach is that the client requires executing two requests to perform the query instead of one as in the OGC standard. Also, there is an overhead to the server for creating one URL representation for each query executed against the server. For queries involving more than one resource, the same approach can be used. However, the proposed JSON serialization for Filter Encoding will have to be extended, being left as a recommendation for future work.

2.4.2 Web Processing Service

The WPS standard provides an interoperable protocol for wrapping any computational process (spatial and non-spatial), specifying the interface for input and output, and indicating how clients can execute the process [7]. In this thesis, we focus on WPS asynchronous execution, in which the client executes the process by a POST operation, the server creates a job, and the client can keep track of the status of the process. Whenever the job is completed, the server returns a URL through which the client can access the results. The advantages of asynchronous execution are that it allows long-running executions without locking the request of the client, it allows the server to manage its resources by starting the process whenever it is most suitable, and (as discussed in Chapter 7) the use of the asynchronous execution can reduce the bandwidth usage in a service composition scenario. As a disadvantage, the client requires more requests to access the results of the process, requiring one POST request to create the job, at least, one GET request to verify whether the job is completed, and the last GET request to retrieve the actual result of the process. Also, the monitoring of the job status may require several requests if the server is not able to provide an accurate ETA of the process; this problem is addressed by the use of WebSockets, as we discuss in Section 2.5.

The RESTful bindings for WPS follow the same principle as described for WFS. It uses three HTTP methods:

- GET, for retrieving information about a job status, requests the results, and to access metadata
- POST, to execute a process
- DELETE, to dismiss a job, as described in the WPS Dismiss Extension.

The resources of WPS are the processes, the jobs, and the outputs of the processes. These resources can be defined in a hierarchical structure as exemplified in Table 2.2.

Table 2.2: Example of resource URL for WPS

OGC KVP	http://www.example.com/wps.cgi?SERVICE=WPS& VERSION=2.0.0& REQUEST= GetResult& jobId =286e8cbd-7d51-48c5-ad72-b0fcbe7cfbdb
REST	**GET** http://www.example.com/wps/2.0.0/processes/buffer/jobs/286e8cbd-7d51-48c5-ad72-b0fcbe7cfbdb/outputs/buffered

The Code-on-Demand constraint can also be applied to RESTful WPS bindings. With Code-on-Demand, a process can be executed on the client side instead of on the server. In many situations, this will increase the performance and the efficiency of the process by not requiring to transfer input data to the server and requesting the output data. Also, it can be applied to reduce the bandwidth usage in a composition of services. This idea was originally introduced for spatial applications in [49] and is out of the scope of this thesis, being left as recommended work.

To harmonize passing data between OGC Web Services, it is desirable that the result of an asynchronous execution of a WPS process behave in the same way as that by a WFS, indistinguishable to the client. This requires that each output of the process be a resource, that has a unique URL and that its metadata is dynamically generated based on the metadata of the process and the

inputs. To identify outputs as resources, we develop the hierarchical structures exemplified in Table 2.2, such that a specific process can be accessed by its unique identifier within the service, in this case, *buffer*, and then the specific job can be accessed by the unique job identifier, and each output has a unique identifier within the process. This resource should be implemented as a WFS, which allows parameters and operations as the application requires. Methods for metadata propagation are discussed in Chapter 6.

2.4.3 Catalogue Service of the Web

The CSW standard defines implementation rules of a catalogue of geospatial records, which provides the means of discovery, browse, and query of metadata about geospatial data and services [8]. The RESTful bindings shall follow the same principle as presented in the previous sections. It uses five HTTP methods:

- GET is used to retrieve one or more records, and to retrieve metadata.
- DELETE is used to delete a specific record.
- PATCH/PUT is used to update a specific record.
- POST is used to insert one or more records, and also to execute the Harvest Operation.

In the Harvest Operation, the user requires providing to the CSW a series of entry point URLs for other OGC services, which the CSW requests the metadata of those services, inserting or updating its records. The records, information that is catalogued, are the only resource defined for this service type. Also, in this thesis, we extend the concept of CSW to allow storage of information other than service metadata. In Chapter 7, CSW is used to store the description of compositions.

To retrieve records effectively, it is necessary that the service can handle complex queries. Since the CSW has only one resource, the query processing is simpler than in WFS. The WCS standard states that the minimum filter capability that should be implemented in the server comprises:

- Logical Operators: and, or, not.
- Comparison Operators, namely property is: equal to, not equal to, less than, greater than, less than or equal to, greater than or equal to, like.
- Spatial Operators: bbox.

Those operators are serialized in XML according to the Filter Encoding Specification [50]. Section 2.6.2 presents a JSON serialization for Filter Encoding that is used in the RESTful services implementation. In the REST architecture, the JSON-serialized query should be sent to the server with a POST operation in the auxiliary resource Stored Queries, and the server shall respond with a Location Header pointing to the URL to obtain the results of the query. Stored Queries are not part of the CSW standard, and are proposed here to enable complex queries compliant to REST architecture. A drawback of this architecture is that the client is required to perform two requests to execute the query (one POST to query, one GET to retrieve the results), and also creates overhead in the server of maintaining unique identifiers for each query that is performed. An advantage of this procedure is that the query result since it has a unique URL, can be reused by the client without submitting a new Filter Encoding payload to the server, or even be cached in case that repetitive use is needed.

Another concern for CSW is to keep the records up-to-date. The OGC standard does not recommend any approach other than indicating that the Harvest operation can be scheduled and executed periodically. In the REST architecture, it would solely involve the CSW to perform multiple requests to the catalogued services to verify if there was any change in metadata. A solution to this problem is presented in the next section.

2.5 WEBSOCKETS

WebSockets is a communication protocol that provides bidirectional communication over a single TCP (Transmission Control Protocol) connection [51]. WebSockets presents a different paradigm compared to the HTTP protocol. HTTP uses the idea of a request, for which the client opens a connection, the data it requested is transferred, and then the connection is closed. Websockets, on the other hand, uses an open and persistent connection, which takes away the overhead created by opening and closing connections. The second paradigm difference is that a bidirectional connection does not only allow browsers to request (pull) data but also allows the server to push data to the browser. This functionality is particularly interesting when a client needs to keep track of things changing on the server. By the HTTP specification, such would require the client to poll the server repeatedly, creating an overhead both on the client and the server. Another advantage of this protocol is that is implemented in all major browsers, such as Chrome, Firefox, and Internet Explorer.

In this thesis, the idea of WebSockets is applied in two different situations. The first is in the communication between the client and the WPS server when tracking the status of a job. When an asynchronous processing is created in WPS, a job URL is created by the server, and the client needs to keep track of whether the job succeeded. The approach recommended in the specification is that the server should estimate the completion of the job (ETA) and inform the client, which will only request again for the status of the job at or after the indicated time. This method reduces the overhead of communication between client and server, however, not all servers can provide an accurate estimate of completion, or can even estimate the process at all. To overcome this problem, the WebSockets protocol can be used. When a job is created, it returns a WebSocket identifier to which the client can open a connection and make use of the bidirectional connection provided by the protocol, which allows the server to inform the client of job status changes.

The second idea is to create a communication between a CSW and other services, to ensure that updates in metadata reach the catalogue. The OGC Specification currently does not recommend any approach to managing the harvesting of metadata to keep the catalogue up-to-date. This can potentially be solved by using WebSockets connections, with a socket for each service to which a client can connect and receive information about updates in the metadata. This socket can be utilized by a CSW, which can monitor all changes to the metadata and request a harvest operation accordingly, and ensure that it has the most up-to-date information. The overhead of keeping open multiple long socket connections is balanced by the low amount of data transferred since the updates in metadata are usually not frequent. This can also be used to verify whether a service becomes unavailable.

Both ideas were implemented in the services and used in the Orchestration Client discussed in Chapter 7. WebSockets can also be potentially used to enable WPS to process stream data, however, this idea is left as recommended work.

2.6 JSON IN OGC STANDARDS

JavaScript Object Notation (JSON) is an open, lightweight, text-based data interchange format [52]. It has six types: String, Number, Boolean, Null, Object, and Array. The JSON format is currently the most used format to exchange data between client and server in the Web, and this is because non-complex (without a high number of nesting) JSON files have a lower payload compared to XML. Also, the browser can parse JSON quicker, being easily integrated with JavaScript code (since JSON object is a JavaScript Object). Lanthaler, 2012 [53] states that the main reason that JSON replaced XML as the most used data format for Web APIs is the X/O impedance mismatch, which is the difficulty of processing XML in an Object Oriented paradigm [54]. Currently, XML/GML is the main data format in OGC services, requiring any server to respond to requests in this format. The OGC services allow other formats to be delivered. However, they do not standardize the requests and responses in those formats. The use of JSON potentially increases the adoption of OGC Standards. This is especially true when using RESTful APIs when the user expects that the application can serve JSON.

Since JSON is simpler than XML, it has several limitations. The first is that JSON does not support complex data types, restricting the use of classes as in GML, which harms interoperability. The second is the lack of a mechanism to define namespaces as in XML, which is used to identify the context in which a property or class is used in XML, and which also resolve naming ambiguities. The third is the lack of a standardized mechanism for JSON validation, such as XML Schema Definition (XSD) for XML, which is a fundamental part of the OGC standards. The fourth is the lack of a query language for JSON such as XPath for XML. This is especially a limitation in OGC standards since many of the operations are based on XPath to reference elements. The fifth disadvantage is the lack of a mechanism to link concepts in JSON, such as provided by XLink in XML.

Two approaches can be cited regarding the use of JSON in OGC Standards. The first is JSON Interfaces for XML (JSONIX), a JavaScript library capable of performing marshalling (JSON to XML conversion) and unmarshalling (XML to JSON conversion). The approach consists of using JSONIX as a middleware between the server and the client that performs the XML conversion. The advantage is that no modification is necessary for the OGC Services. As a disadvantage, this creates an overhead of marshalling every client request with a JSON payload and of unmarshalling every server response, which can be time-consuming for large geometric objects. The second approach turns JSON into a primary data format for OGC services as described in Testbed 11 [24]. This document discusses the challenges of using JSON in the OGC standards and provides a series of recommendations. It is used as the basis for this thesis in applying JSON to the proposed RESTful OGC services both for data encoding and metadata, and our work complements that of Testbed 11 in the following areas:

- Definition of JSON serialization for Filter Encoding.
- Validation of JSON objects using JSON-Schema.
- Alternative for XLink using JSON-LD.
- Use of JSON-LD to handle namespace definitions such as in XML.
- Use of JSON-LD to define complex data types.
- Standard way of representing spatial features with an extension of GeoJSON to JSON-LD.

In the next sections, we introduce GeoJSON as a format for sharing spatial data within the proposed services, discuss the proposed JSON Serialization for Filter Encoding, and discuss the

use of JSON-Schema for validation of the JSON file. The extension of JSON to JSON-LD is addressed in Chapter 6.

2.6.1 GeoJSON

GeoJSON is a JSON-based open format, currently in the standardization process by the Internet Engineering Task Force (IETF), designed to represent geometries, features, and collection of features. GeoJSON supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. For each GeoJSON feature, one can associate a geometry, a group of properties, a coordinate reference system, and a bounding box [55].

GeoJSON adoption in the Web is increasing, especially due to the ease of use combined with mapping libraries such as Leaflet and OpenLayers. OGC already developed standards based on GeoJSON encoding such as the proposed standard OWS Context with GeoJSON Encoding [56]. Since GeoJSON is the natural way of encoding geospatial features in JSON, we are adopting it in this thesis.

The disadvantages of GeoJSON are: the JSON format is not as efficient as binary encodings such as Shapefile or WKB; it can only have one geometry associated with the feature, as opposed to GML, which can have multiple geometries; properties are restricted to JSON data types; it cannot represent all geometries of the Simple Features Access Standard [57], such as curves; there is no mechanism to restrict geometry types in a Feature Collection, meaning that it can be a heterogeneous collection of geometries, adding an extra overhead in the validation; and lastly, GeoJSON features and geometries have no semantics associated, meaning, for example, that one has no means of identify what a feature represents a road or that the geometry represents the center of the road or the margins. To be consistent for use in WFS, it is at least required a modification of GeoJSON in such a way that each feature can provide a property which specified the feature type.

In this thesis, GeoJSON is used both for providing spatial data and also in metadata fragments that contain a geospatial object. In Chapter 6, GeoJSON is extended with JSON-LD principles that allow providing semantically enabled geospatial data. Also, some of the disadvantages of GeoJSON are addressed, which allows a more consistent implementation of GeoJSON in the OGC Standards.

2.6.2 JSON Serialization for Filter Encoding

Filter Encoding is necessary for the definition of Catalogue Services for the Web (CSW) and optional in the use of Web Feature Services (WFS). It is standardized in OGC Filter Encoding 2.0 Standard and ISO 19143 Filter Encoding. In this section, we describe a JSON Serialization for Filter Encoding (JSFE) operations with a notation based on MongoDB [58] query parameters. It uses GeoJSON geometry encoding, in contrast to OGC Filter Encoding that uses GML geometries. In Appendix A, we describe with more details the JSON Serialization for Filter Encoding (JSFE).

JSFE presents three types of operators: Logical Operators, Comparison Operators, and Spatial Operators. Each operator is defined as a key in a JSON object with a prefix $ and is written in lowerCamelCase, as opposed to the Filter Encoding standard that specifies the names of the operators in UpperCamelCase. JSFE is expected to have a lower payload compared to Filter Encoding, is easier to parse, it allows validation by JSON-Schema (discussed in the next section), and to follow JSON format, having a consistent format across applications. Currently, JSFE is not able to express Temporal Operators, to encode queries that involve more than one feature layer, and it also does not provide ad-hoc query capabilities. The extension to involve more than one feature

layer is not trivial since OGC Filter Encoding relies on XPath expressions, for which a standard alternative for JSON is hard to define.

Table 2.3 presents a comparison between XML and JSON serialization for the Filter Encoding of an operation to find buildings of type Police Station or Fire Brigade within 2000 meters of a point.

Table 2.3: Comparison between OGC Filter Encoding and JSON Serialization

Filter Encoding	JSON Serialization

`<And>`
`  <Or>`
`  <PropertyIsEqualTo>`
`  <PropertyName>`*type*`</PropertyName>`
`  <Literal>`*Police Station*`</Literal>`
`  </PropertyIsEqualTo>`
`  <PropertyIsEqualTo>`
`  <PropertyName>`*type*`</PropertyName>`
`  <Literal>`*Fire Brigade*`</Literal>`
`  </PropertyIsEqualTo>`
`  </Or>`
`  <DWithin>`
`  <PropertyName>`*geom*`</PropertyName>`
`  <gml:Point>`
`  <gml:coordinates>`*125.6 10.1*
`  </gml:coordinates>`
`  </gml:Point>`
`  <Distance>`*2000*`</Distance>`
`  </DWithin>`
`</And>`
	`{`
	`  $and: [`
	`  {$or: [`
	`  {type: {$eq: "Police Station"}},`
	`  {type: {$eq: "Fire Brigade"}}`
	`  ]},`
	`  {geom: {$dWithin:`
	`  {geometry: {type: "Point",`
	`  coordinates: [125.6, 10.1]},`
	`  $distance: 2000}}}`
	`  ]`
	`}`

2.6.3 JSON Schema

Schema validation is an important part of the OGC standards and so in our approach, we need a mechanism that mimics XSD validation when using JSON. JSON Schema is a JSON-based, human and machine-readable description of a JSON file [59]. It aims to validate the expected JSON keys and corresponding types, which allows the verification and documentation of the data model. As opposed to the information stated in Testbed 11 [24], even with the fact that JSON objects are extensible by default, JSON Schema still provides means of restricting additional attributes that are not described in the Schema, which allows a verification as strict as XSD on this criterion. One point in which validation differs is that the property order in a JSON object cannot be validated, which in the OGC standards' scope is not a problem since no restrictions are imposed regarding this matter.

As advantages of JSON Schema, we should mention that it can be extended, which allows the verification of constructed types other than the basic JSON types. Also, JSON Schema uses JSON syntax, so can be parsed and validated quickly by the browser. As a disadvantage of JSON Schema, we should mention that it is not an official standard, and the IETF Draft expired in 2013, meaning that enforceability of this specification is low. In the case of adoption by OGC, it requires continuation of the standardization process. Another disadvantage is that the validation

of JSON-LD by JSON-Schema restricts the flexibility of JSON-LD since one can represent the same information using different structures. This can be solved by the user receiving the payload first to apply JSON-LD Framing to bring the object into an expected structure and then applying a JSON-Schema verification. This solution needs to be compared to a more natural solution of converting JSON-LD to RDF, validate the RDF graph, and serialize back RDF to JSON-LD.

In this thesis, we adopt JSON-Schema as the means of validating JSON in the proposed OGC services. The validation occurs when parsing payloads of POST requests, verifying if the feature being inserted/updated, or the process being submitted is valid. Schemas used to validate GeoJSON are adapted from [60]. They are also modified to validate properties from JSON-LD and are reorganized so that specific validations, such as restricting one geometry type, can be performed.

2.7 SUMMARY

In this chapter, we discuss how OGC services can be modified following current web technology and the trade-off of applying this change to the architecture. We propose a modification of the standards so that RESTFul bindings can be implemented, providing a simpler alternative to the KVP and SOAP bindings. The use of REST architecture provides scalability to the server, and also renders the services lightweight compared to RPC/SOAP/XML implementations, especially when combined with JSON as the main data format. The proposed services are flexible since they allow an individual definition of methods and parameters for each resource, and as the application needs. One notable change in the WPS service is that the result of an asynchronous execution is indistinguishable from a WFS layer, which harmonizes data passing between WFS and WPS.

We provided two scenarios in which the WebSockets protocol is applied in the OGC Specifications, complementing the cases where REST architecture leads to an inefficient implementation. WebSockets can be used to enhance the capabilities of a CSW to keep metadata up-to-date, and lowers the overhead of communication when tracking the progress status of a job in an asynchronous execution of WPS. Lastly, we discussed the use of JSON as a data format for sharing metadata and spatial data in OGC services. We addressed the limitations of GeoJSON for sharing spatial data compared to GML and proposed solutions with JSON Schema and JSON-LD.

From here on in the thesis, whenever OGC services are mentioned the reader should consider that these are the OGC Services modified in the way described in this chapter. Chapter 3 discusses how those services can be composed, which possible forms of service composition can be recognized, and it provides theoretically founded methods to verify its validity.

Chapter 3
Theory of Composability

3.1 INTRODUCTION

In Chapter 2, we defined the web services of interest in this thesis: WFS, WPS, and CSW. In this chapter, the focus will be on how data providing services (such as WFS) and data processing services (such as WPS) can be composed in a valid and meaningful way. ISO 19119 [61] defines web service composition as a sequence of web services in which the results of one service are necessary for the execution of a second service. In this thesis, service composition refers to the process of selecting and assembling a meaningful combination of services to solve a specified problem that cannot be solved by individual services. Composability is the characteristic of a particular composition not result in execution errors. An *execution error* can be defined as an incorrect step that causes a service to behave in an unintended or not anticipated manner. Two types of execution error can be distinguished: *trapped errors*, which makes the execution stop immediately; and *untrapped errors*, which are not caught by the service, later causing arbitrary behavior in the output. Theoretically founded service composability allows describing precisely in which cases a number of web services can be composed as sequential processes, producing valid information.

The verification of composability can be performed at compile-time, before the execution of the composition, or at run-time, while the composition is executing. In this thesis, the focus is compile-time verification. Nevertheless, all the proposed mechanisms can be applied at run-time. Compile-time verification is especially important in the composition of geospatial web services due to the computational complexity of spatial functions that can lead to long running times in each operation, as well taxing use of the server processing capabilities. In this way, an early identification of errors and prevention of composition execution can save time, processing capabilities, as well as lead to a more robust composition. Also, since the composition is executed in a distributed environment over the Web, the verification before execution can remove unnecessary data transfer. The drawback is that not all errors can be verified at compile-time, as we further discuss in Chapters 4 and 5.

In this chapter, we discuss which forms of composition can be recognized, and for each form how far we can decide on its validity. Section 3.2 exposes the previous approaches in verifying the validity of service composition. Section 3.3 divides the composability problem into levels, which are verified in a sequential manner. Finally, Section 3.4 presents the first level of composability, which abstracts the composition as a graph, defining invalid interactions in a composition, as well as mechanisms to identify those cases.

3.2 PREVIOUS APPROACHES

There is a lack of literature specific to geospatial service composition verification [62]. All the approaches found are based on the W3C WS-* family of web services, which are based on technology such as Web Service Description Language (WSDL), and Business Process Execution Language (BPEL).

Rusli et al. [63] conducted a systematic literature review in the field of testing web services composition. In this literature review were used 42 papers, which identified that all approaches used BPEL or Web Ontology Language for Services (OWL-S) for describing the web service composition. Also, 26 of the 42 papers focused on test generation for verifying web service composition. In this thesis, a formal theory is introduced to describe the valid interaction between web services, not requiring the generation of test cases.

Milanovic, 2005 [64] recognizes the need for a formal approach to verify composition correctness and it defined a Contract Definition Language based on Abstract Machines which allows the description of preconditions, postconditions, and invariants of web services. The motivation for the development of this language is that approaches based on BPEL and OWL-S lack means of verifying composability. The drawback of this work is that the specified preconditions and postconditions are not sufficient to ensure correctness of the composition, still needing to cover semantic and structural aspects. Also, the work does not discuss how the different patterns of input and output interactions interfere in the verification of pre and postconditions.

Baryannis et al. [65] provides a mechanism for deriving specifications for composite web services based on descriptions of their components. The resulting specification formally describes in First Order Logic the composition based on inputs, outputs, preconditions, and effects. The following interactions between services are discussed: sequential composition, AND-split, OR-split, XOR-split, and loop. The drawback of this work is that the approach does not take into account input/output interaction, just service interaction. Also, this work did not implement the proposed mechanism, creating doubt about its applicability.

Chen et al. [66] uses a method based on a Type Theory with the goal of achieving automatic service composition. It is based on Martin-Löf type theory, and a subtyping mechanism is used if an output of a service can become the input of the next. The drawback of the approach is that is not discussed parametric types, and consequently, the output types are static, not depending on the type of the provided input; this reduces considerably the expressiveness of the check. In this thesis, a mechanism for type propagation is developed to define outputs types based on the input types.

Kim et al. [67] proposes a framework for automatic service composition. It divides the composition process into five phases: specification, planning, validation, discovery, and execution. In the validation phase is divided into syntactic validation, which verifies the structure of the composition, restricting deadlocks, infinite cycles, and others; and semantic validation, that verifies the composition against the user requirements. However, this work does not describe methods for the verification of a composition.

Medjahed and Bouguettaya [68] proposes the verification of composability based on rules. Those rules are organized into four different levels: syntactic, static semantic, dynamic semantic, and qualitative. The goal of the framework is to calculate a composability degree, which provides a quantitative measure of how well composed are the services. In this thesis, this measure is nonsensical, since the concept of composability is tightly coupled with the idea of the absence of execution errors (trapped or untrapped). In this way, compositions that are not valid under the rules presented in this thesis should not be executed, since it produces an execution error.

3.3 LEVELS OF COMPOSABILITY

Following the ideas of [67] and [68], in this thesis the verification of composition is divided into five levels of composability based on the expressiveness of the checking, namely: Structural, Static Syntactic, Dynamic Syntactic, Semantic, and Qualitative. Each of the levels gives a definition to service and service composition and specifies which type of errors can be identified, and how to

verify the existence of those errors. Figure 3.1 exposes the defined levels of composability.

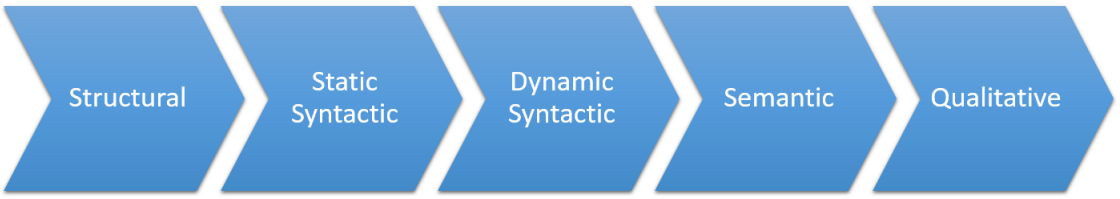


Figure 3.1: Levels of Composability

Structural Composability abstracts the composition as a directed graph, and defines in which conditions this graph is well-formed. In this graph services are represented as nodes and edges as the passing of data between an output and input. This verification does not take into account specifics of each service, the goal being to test for invalid connections such as cycles, duplicates, verify the existence of required connections, and validity of loop. We discuss this level in Section 3.4.

Static Syntactic Composability states that the output of a component service can only become the input of the next service if their data types are identical or the output data can be automatically coerced to the data type of the input parameter. We desire to verify this characteristic at compile-time with the goal of preventing a class of execution errors. At this level, we only verify whether the data type of the input data matches the signature of the service input. In this thesis, geospatial web services and service composition are seen as governed by the rules of typed functions, in which spatial data types play a fundamental role to characterize inputs and outputs, and consequently service types. A type checking algorithm is developed to verify the existence of invalid interactions. We discussed the type check rules in Chapter 4 and the implementation in Chapter 7.

Dynamic Syntactic Composability complements Static Syntactic Composability by covering restrictions related to data values that cannot be expressed in the type system. These include interaction between inputs, coordinate systems, geometry dimensions, time zones, and pixel size. In this thesis, the restrictions in the data are formalized as preconditions in Hoare Logic, which was extended to handle explicitly inputs and outputs. Also, since the focus of this thesis is compile-time verification, the information for the verification of this level of composability is not based on the data, but on the service metadata, which provides an abstraction of the possible outcome of the service. It is provided a formalization about how a service can access this metadata information, and how the pre- and postcondition can be chained, allowing the evaluation of a composition. During run-time, the verification can be executed based on the data. We discuss this level in Chapter 5.

Semantic Composability addresses the meaning of data and processes, verifying if the combination of processes can derive a meaningful result, or if it produces the intended result of the user. Services are considered to be semantically enabled, having semantic metadata associated with inputs, outputs, and with the service functionality in general. In this thesis, we developed a mechanism to identify attributes which represent the same relationship and normalize their names based on the context provided by the service. Also, we discuss the use of geooperators thesaurus to classify web services, which allows the development of a mechanism for service substitutability, and potentially the suggestion of services. We discuss this level in Chapter 5.

The last level is *Qualitative Composability*, the evaluation of the composition against user requirements related to non-functional characteristics such as response time, service availability, cost, security, legal rights, and quality of operations. This level of composability is not discussed in this thesis since it fundamentally is not related to protection against errors, but to the best selection of services to fulfill the user specification.

In the next section, we discuss the first level of composability, Structural Composability.

3.4 STRUCTURAL COMPOSABILITY

In Structural Composability, we abstract the composition as a directed graph and define in which conditions this graph is well-formed. A graph is a mathematical structure that represents objects and the pairwise relationship between them. The object is called *node*, and the relationship *edge*. For our scope, we define three disjoint classes of nodes: the service node, the input node, and the output node. If exists an edge between two nodes we say they are *connected*. Let n and m be nodes, the edge connecting the node n to m is denoted (n, m). The edge is directed, meaning that $(n, m) \neq (m, n)$. Next, we give a definition of a service and a service composition based on the concepts introduced so far.

Definition 3.1 (Service)**.** A *service* is a set containing one service node, zero or more input nodes, and one or more output nodes, such that exists edges connecting the service node to each of the input and output nodes, and there are no edges connecting inputs and outputs nodes.

Definition 3.2 (Service composition)**.** Let N_s be the set of all nodes of type service in a graph, N_i the set of all nodes of type input, N_o the set of all nodes of type output. A *service composition* is a graph $G = (N, E)$, where $N = N_i \cup N_o \cup N_s$ is the set of all nodes in the graph, and E is the set of edges such that:

$$
\begin{aligned}
E &= E_{so} \cup E_{si} \cup E_{oi} \\
E_{so} &\subset N_s \times N_o \\
E_{si} &\subset N_s \times N_i \\
E_{oi} &\subset N_o \times N_i \\
E &\subset N \times N
\end{aligned}
$$

Meaning that E_{oi} are the edges connecting an output node to an input node, E_{so} are the edges connecting a service node to output node, and E_{si} connecting to an input node. This definition implies that connections between a pair of service nodes, output nodes, or input nodes, are forbidden. Also, a composition graph is also restricted to be a simple graph, meaning that between two nodes can only exist a single edge.

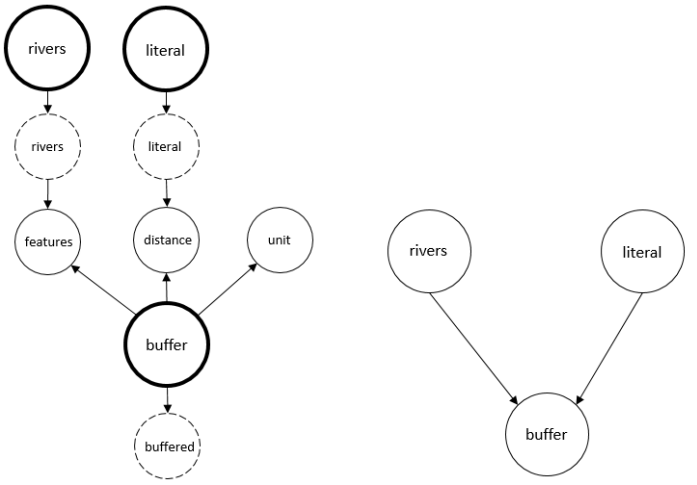
The edges of a service composition have two different semantics, the first, for edges in E_{so} and E_{si}, is the relationship that the input/output belongs to the service; the second, for edges in E_{oi}, is the passing of data between an output of one service to the input of the next.

A service composition is not a labeled graph since nodes do not necessarily have a unique identifier because a service may be needed multiple times. A weaker concept than labels is introduced: tags. A *tag* allows the identification of a service, input and output, however, it is not necessarily unique. We can formalize a tag in the following way:

Definition 3.3 (Tag)**.** A *tag* is a string associated with a node in the composition graph. The set of all tags is denoted T. The function *tagValue* associates a node with its tag:

$$\text{tagValue} : N \to T \tag{3.1}$$

Figure 3.2 shows the graphical representation of a graph. In the complete graph representation weighted circles denote services nodes, small dashed circles denote input nodes, and the remaining



(a) Complete graph representation with inputs and outputs

(b) Simple graph representation

Figure 3.2: Comparison of graph representations

denote output nodes. Also is presented the simple graph representation, that omits input and output nodes.

In the next section, we formalize the conditions that a service composition is valid.

3.4.1 Composition Well-formedness

We defined a composition as *well-formed* if its execution does not produce structural errors. In this section, we define the possible structural errors.

The first structural error is dependency cycles. A *dependency cycle* is a pair of services that either directly or indirectly depend on each other. The reason for this restriction is straightforward since it will not allow the complete execution of the composition, entering in an infinite queue state waiting for a result that cannot be derived. This means that the composition must be a Directed Acyclic Graph (DAG). To formalize cycles we first introduce the notion of a service path.

Definition 3.4 (Service path)**.** A *service path* between the service nodes a and b is a sequence of service nodes $[s_0, \ldots, s_k]$ such that $s_0 = a$ and $s_k = b$ and:

$$\begin{aligned} \forall j \in [0, 1, \ldots, k-1] \mid \exists i \in N_i \mid \exists o \in N_o \mid (s_j, o) \in E_{so} \wedge \\ (s_{j+1}, i) \in E_{si} \wedge \\ (o, i) \in E_{oi} \end{aligned} \tag{3.2}$$

Definition 3.5 (Dependency cycle)**.** A *dependency cycle* is a nonempty service path where the start and end nodes are the same.

To verify the existence of cycles we use the Tarjan's strongly connected components algorithm [69], which operates on a directed graph, and produces a partition of the graph into strongly connected components, meaning that for every pair of nodes x and y inside a component there is a directed path that connects x with y and y with x. In this way, a graph is acyclic if, and only if, the result of Tarjan's algorithm are graph partitions with only one node as a component.

The next structural error is the restriction in the name of inputs and outputs within a service. To keep consistency in the service signature, it is required that all input tags that are connected to one service must be different. The same apply to output tags.

Definition 3.6 (Unique tags)**.** For each service, the tags of inputs connected to it must be distinct. The same apply to output nodes.

$$\begin{aligned} \forall s \in N_s \mid \forall o_1 \in N_o \mid \forall o_2 \in N_o \mid (s, o_1) \in E_{so} \wedge \\ (s, o_2) \in E_{so} \wedge \\ o_1 \neq o_2 \\ \Rightarrow \text{tagValue}(o_1) \neq \text{tagValue}(o_2) \end{aligned} \tag{3.3}$$

$$\begin{aligned} \forall s \in N_s \mid \forall i_1 \in N_i \mid \forall i_2 \in N_i \mid (s, i_1) \in E_{si} \wedge \\ (s, i_2) \in E_{si} \wedge \\ i_1 \neq i_2 \\ \Rightarrow \text{tagValue}(i_1) \neq \text{tagValue}(i_2) \end{aligned} \tag{3.4}$$

Services are also restricted not to have a self-connection, meaning that an output input edge cannot link a pair of input/output that are linked to the same service, which creates a nonsensical structure.

Definition 3.7 (Self connection)**.** Cannot exist an edge connecting input and output edges connected to the same service edge.

$$\begin{aligned} \forall s \in N_s \mid \forall i \in N_i \mid \forall o \in N_o \mid (s, i) \in E_i \wedge \\ (s, o) \in E_o \Rightarrow \\ (o, i) \notin E_{oi} \end{aligned} \tag{3.5}$$

Another restriction that is imposed on the composition graph is that it requires to be weakly connected, meaning that from any starting service node is possible to reach any other service node by traversing the graph from output to input, or input to output. This simplifies the graph to a unique overall process and implies that if two independent groups of tasks are being performed on the same graph, they need to be broken into two graphs.

Definition 3.8 (Undirected service path)**.** An *undirected service path* between service nodes a and b is a sequence of service nodes $[s_0, \ldots, s_k]$ such that $s_0 = a$ and $s_k = b$ and:

$$\begin{aligned} \forall j \in [0, 1, \ldots, k-1] \mid \exists i \in N_i \mid \exists o \in N_o \mid ((s_j, o) \in E_{so} \wedge (s_{j+1}, i) \in E_{si} \wedge (o, i) \in E_{oi}) \vee \\ ((s_{j+1}, o) \in E_{so} \wedge (s_j, i) \in E_{si} \wedge (o, i) \in E_{oi}) \end{aligned} \tag{3.6}$$

Definition 3.9 (Weakly connected graph)**.** A *weakly connected graph* is a graph in which for every pair of distinct service nodes exists a nonempty undirected service path.

The last two structural errors relate to two additional conditions that an input node can have, the required and the unique condition. The *required condition* states that the input node must be connected to at least an output input edge. The *unique condition* states that the input node must be connected to at most one output input edge.

Definition 3.10 (Required input)**.** Consider the following function relating input nodes and a boolean:

$$\text{required} : N_i \rightarrow \texttt{boolean} \tag{3.7}$$

A input node i is defined *required* if $required(i) = true$. And for required nodes the following applies:

$$\forall i \in N_i \mid \exists o \in N_o \mid \text{required}(i) \Rightarrow (o, i) \in E_{oi} \tag{3.8}$$

Meaning that in the composition graph it is necessary to exist an edge in E_{oi} connected to that node.

Definition 3.11 (Unique input)**.** Consider the following function relating input nodes and a boolean:

$$\text{unique} : N_i \rightarrow \texttt{boolean} \tag{3.9}$$

A input node i is defined *unique* if $unique(i) = true$. And for unique nodes the following applies:

$$\forall i \in N_i \mid \forall o_1 \in N_o \mid \forall o_2 \in N_o \mid \text{unique}(i) \Rightarrow ((o_1, i) \in E_{oi} \wedge (o_2, i) \in E_{oi} \Rightarrow o_1 = o_2) \tag{3.10}$$

Meaning that in the composition graph it is necessary to exist at most one edge in E_{oi} connected to that node.

Remark. Prior to any verification of structural composition the composition representation requires being syntactically validated against the capacity of the orchestration engine. This includes the validation of the format that represents the composition, verify whether the data access binding of the services is supported, whether the transferred data file type is supported by the service. As this validation is specific to each orchestration engine, it is not covered by the theory. However, the specific checks for the orchestration engine implemented in this thesis are discussed in Chapter 7.

Figure 3.3 presents examples of structural errors.

3.4.2 Conditional Nodes

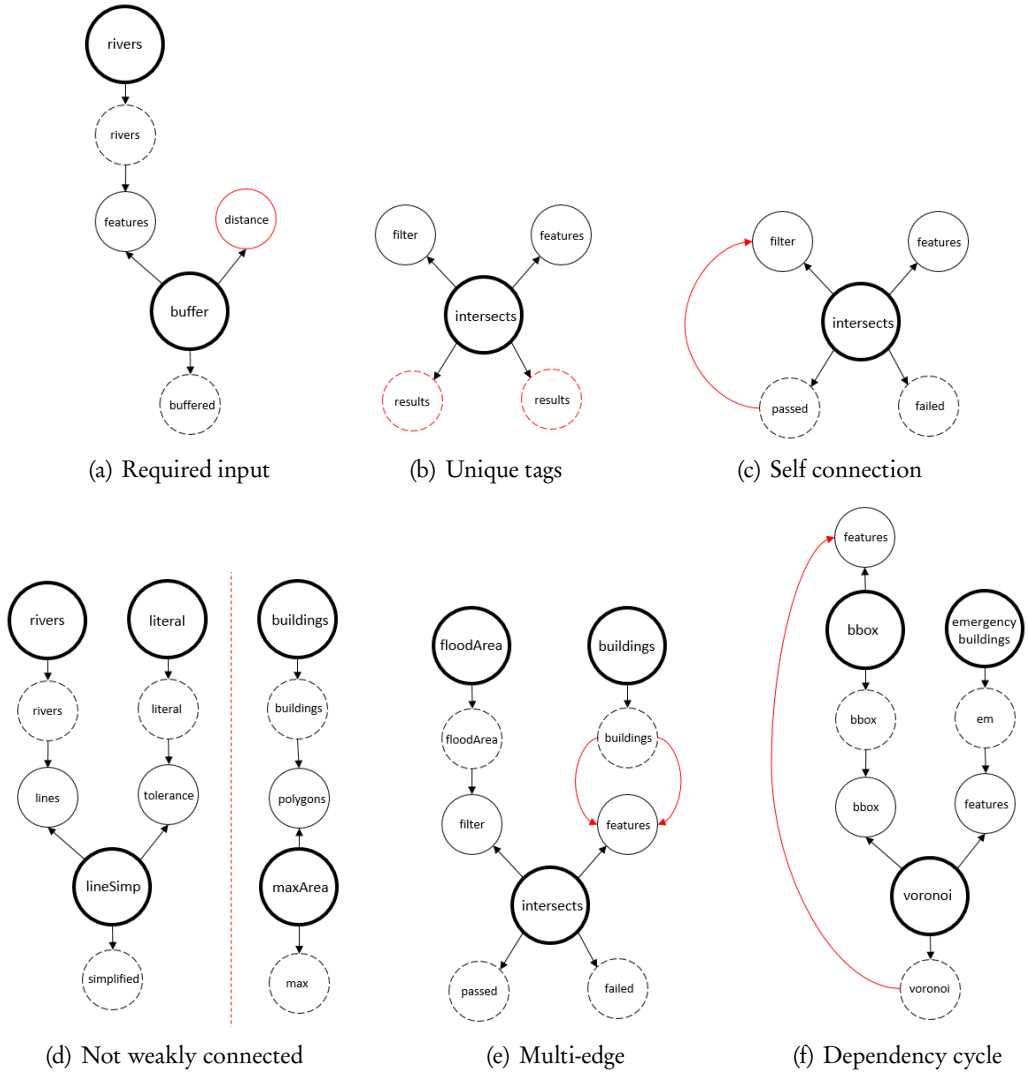
In this section, we introduce one more type of node: the conditional node. A *conditional node* allows a boolean verification of an expression over input sets at run-time. Based on the result of the verification it provides two outputs, the *true* output if the condition evaluates to true, and the *false* output if it evaluates to false. This node is an exclusive disjunction, meaning that the composition will only use one of the outputs, which branches the composition into two different scenarios.

Definition 3.12 (Conditional node)**.** A conditional node is a type of service node, with one input of tag *input*, and two outputs with tags *true* and *false*. Let N_{cond} be the set of all conditional nodes, the following apply:

$$N_{cond} \subset N_s$$
$$\forall n \in N_{cond} \mid \forall i \in N_i \mid (n, i) \in E_{si} \Rightarrow \text{unique}(i) \wedge \text{required}(i)$$

We cannot directly evaluate well-formedness of a composition with one or more conditional nodes. Based on each possible result of the conditional statements, one graph must be derived. Each derived graph is defined as a *scenario*, and they are ensured not to contain a conditional node. A composition with conditional nodes is well-formed if all scenarios are well-formed.

To generate all the possible scenarios, we develop a *branching algorithm*. It receives as an input a graph with one or more conditional nodes, and it returns a set of scenarios. The pseudocode for this algorithm is presented in Appendix B. It is divided into two steps: mark and clean. In the *mark*



(a) Required input (b) Unique tags (c) Self connection

(d) Not weakly connected (e) Multi-edge (f) Dependency cycle

Figure 3.3: Examples of structural errors

step one conditional node is chosen and all the nodes dependent on the *false* output are marked as 0 and the ones dependent on the *true* output are marked as 1. In the case that a node is dependent on both outputs it is not marked. The *clean* step generates two graphs, one by removing the nodes marked with 0, and one by removing the mark nodes marked with 1. Also, the conditional node is removed in each of the graphs and the edges fixed accordingly. Both steps are applied recursively until all the conditionals are eliminated.

Figure 3.4 shows the result of the branching algorithm applied to a graph with two conditional nodes. We use the simple graph representation; the diamond represents the conditional node.

3.4.3 Loop Nodes

When building compositions, in many situations, there are repetitive series of operations across the composition, or even over different compositions. The repetitive parts can be identified, aggregated, and replaced by a node, which allows reusability of series of operations in different parts of a composition or different compositions. Subgraph is defined in this thesis as an abstraction of

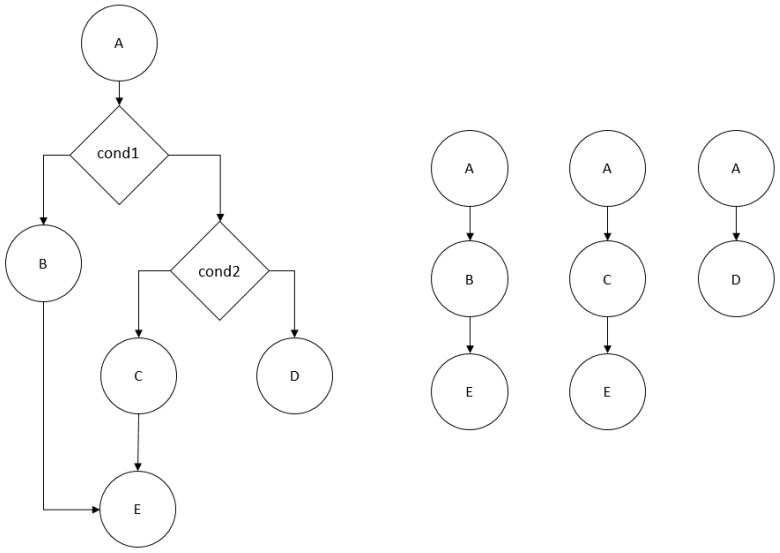


Figure 3.4: Simple graph representation with two conditionals and the possible scenarios

a series of nodes and edges into a single node, the *subgraph node*.

To formalize this, we first introduce three concepts: input parameter node, output parameter node, and parametric graph. *Input parameter node* is a type of service node that is required to be connected to a single output node, and not to be connected to any input node. A similar definition applies to *output parameter node*, as a type of service node, which is required to be connected to one input node, and not be connected to any output node. A *parametric graph* is a type of graph which contains, at least, one input parameter node and one output parameter node.

We define a subgraph node as a type of service node that can be associated with a parametric graph. Let G_p be the set of all parametric graphs, and N_{sub} be the set of all subgraph nodes. We formalize the notion of subgraph as a function that relates a subgraph node and a parametric graph.

$$\text{subgraph} : N_{sub} \rightarrow G_p \tag{3.11}$$

Also, we define an *expand* operation, which replaces a subgraph node with the correspondent parametric graph, fixing the edge connections accordingly. This operation can be performed at compile-time or run-time.

Figure 3.5 exemplifies a subgraph. Is presented the graphic representation for subgraph node (weighted dashed circle), and for input and output parameters a dashed circle. The subgraph node represents the parametric subgraph in the box (not part of the composition graph). The parametric graph is presented in simple graph notation.

To maintain the integrity of the graph, self-referencing is not allowed (the call of the same subgraph node inside a subgraph). This self-referencing can be direct, the subgraph calls itself, or indirect, nested subgraphs that form a cyclic reference. Next, we formalize both cases:

Definition 3.13 (Self reference)**.** A graph is said to have a *self-reference* if:

$$\exists n \in N_{sub} \mid n \in \text{subgraph}(n).N \tag{3.12}$$

Meaning that the subgraph node associated with a parametric graph is a member of the nodes of this parametric graph.

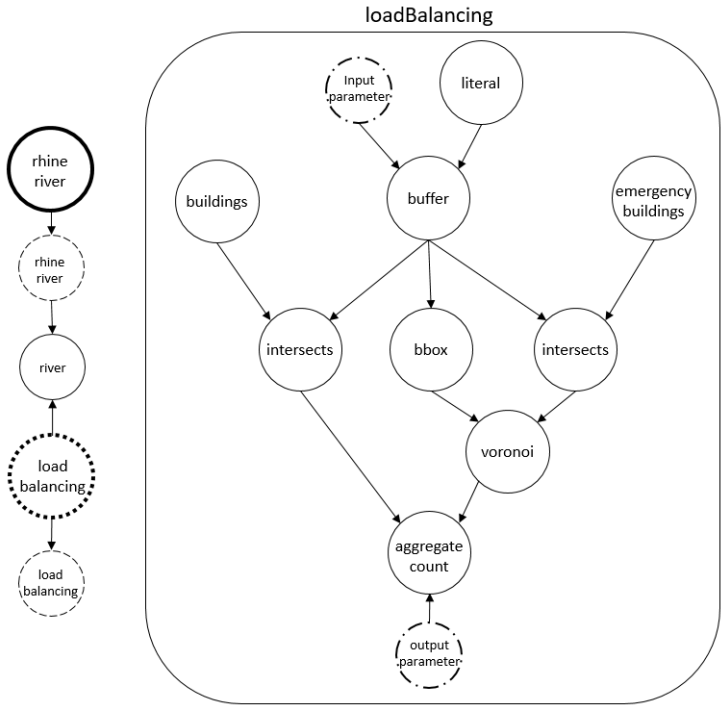


Figure 3.5: Subgraph in simple graph representation

Definition 3.14 (Cyclic reference)**.** A graph is said to have a *cyclic reference* if exists a sequence of subgraph nodes $[s_0, \ldots, s_k]$ such that $s_0 = s_k$ and :

$$\forall j \in [1, \ldots, k] \mid s_k \in \text{subgraph}(s_{k-1}).N \tag{3.13}$$

The concept of a subgraph is further explored in Chapter 6. Based on the definition of a subgraph we can define Loop nodes.

The concern of Structural Composability with a loop is to ensure that the overall composition will eventually halt. A composition design error that leads to an infinite loop in a distributed environment can lead to disastrous consequences, such as overload of the processing services. In this way, it is necessary to ensure that the loop will halt prior to its execution, or have limits on the number of iterations that a loop can perform. The latter is not ideal since a forced stop means that the composition did not complete the intended operation, and can produce unreliable results. The verification whether a program will halt or do an infinite loop, based on the input and description of the loop is known as the Halting Problem [70]. This problem is known to be undecidable, meaning that there exist algorithms for which we cannot prove that they will stop (even though they may do). In this way, to ensure termination, it is necessary to reduce the expressive power of our loop mechanism. We introduce constrained loop that is guaranteed to halt.

We discuss three types of loop: Iterate Inputs, Iterate Sets, and Iterate Multivalue. These are simple loop nodes that are of common use in geospatial science and are defined in a way that guarantees to halt. For each of these, a node type is introduced that behaves in a similar way to a subgraph node. Below, we define each loop type and is given an example.

Iterate Sets receives one input of type Set and performs a series of operations independently on each member of the set. After performing all operations, the result is aggregated back as a set type, and this set is provided as the output. The number of executions of the subgraph is not known at compile-time, but it is known to be finite since it is the same as the number of elements in the

set. The Iterate set node is a type of service node that can be associated with a parametric graph. Unlike the subgraph node, the iterate input node cannot be expanded, since it represents multiple copies of the parametric graph. The parametric graph associated is constrained to have one input parameter node and one output parameter node. The iterate sets node is connected to one input node and one output node. The input node is required and unique.

Example 3.1. Consider the task of calculating a disaster response load balancing (as exemplified in figure 3.5) for a set of major rivers. It receives as input a particular river, and it calculates the affected buildings by a flood in that river, providing the number of buildings that each emergency center would have to support. The load balancing can be defined as a subgraph, and each of the major rivers is passed individually to the function by the use of Iterate Set. A subgraph which passes an individual river to the load balancing subgraph is associated with the node Iterate Set. For each of the features in the input set the function is executed, then the results are grouped together in a set and given in the *results* output. Figure 3.6 shows the graph representation of the multiple executions of load balancing.

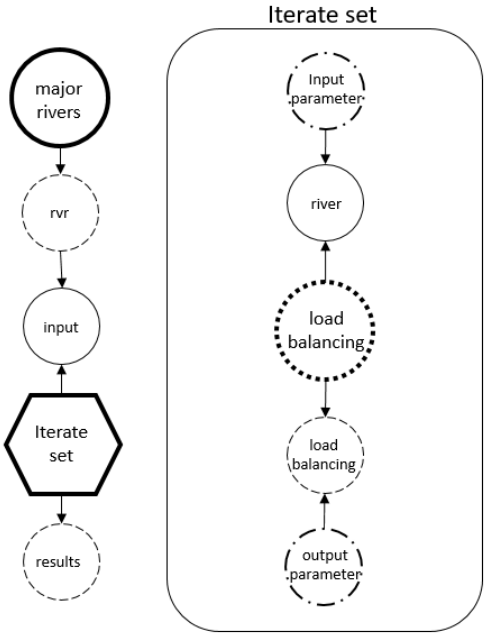


Figure 3.6: Example of Iterate Set

⌟

Iterate Multivalue receives the elements that will be processed and one array of values. For each value in the array, a series of operations is performed on the input. When all values of the array have processed the results for each value are aggregated as one output. For each loop execution, one element of the array is available as an input parameter, in such a way that is performed the same number of loop executions as the number of elements of the array, being known at compile-time. The iterate multivalue node is a type of service node that can be associated with a parametric graph. However, it cannot be expanded. The parametric graph associated is constrained to have two input parameters node, one for the elements of the array of values and one for the input that will be processed, and one output parameter node. The iterate multivalue node is connected to two input nodes, one for the array of values and one for the elements that will be processed, and one output node. Both input nodes are required and unique.

Example 3.2. Consider the task of calculating multiple buffers with specific distance values for a set of rivers. This can be done by applying multiple buffer operations specified with distinct distance values, or it can be simplified by using iterate multivalue. A subgraph of the buffer operation is associated with the node iterate multivalue, the special nodes input parameter and output parameter are used to indicate the passing of data respectively from and to the iterate multivalue node, and the value node is used to pass the distance values. In each iteration, one value of the array of values is passed in the value node, and all the results are aggregated into a single output. Figure 3.7 shows the graph representation of the multiple buffering.

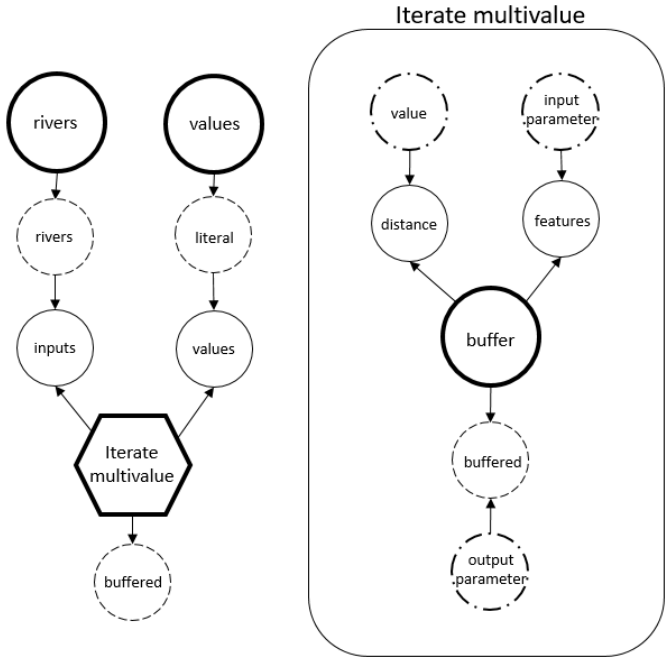


Figure 3.7: Example of Iterate Multivalue

⌟

Iterate Input receives multiple inputs connections and performs a series of operations independently for each of the inputs. Iterate input is a type of service node that can be associated with a parametric graph. However, it cannot be expanded. The parametric graph associated is constrained to have one input parameter node and one output parameter node. The iterate input node is connected to one input node, and the number of output nodes is the same as the number of edges connecting the input node. The input node is required and not unique. The number of operations performed by the iterate input is known at compile-time since it will perform a predetermined operation once for each of the inputs. This node is particularly useful when large parametric subgraphs are used multiple times in a graph.

Example 3.3. Consider the task of clipping multiple feature layers with one layer *areaOfInterest*. This can be done by applying multiple clip operations, or it can be simplified by using Iterate Input. A subgraph of the clipping operation is associated with the node Iterate Input, the special nodes Input Parameter and Output Parameter are used to indicate the passing of data respectively from and to the Iterate Input node. For each one of the inputs, an output is generated. Figure 3.8 shows the graph representation of the multiple clipping operations.

⌟

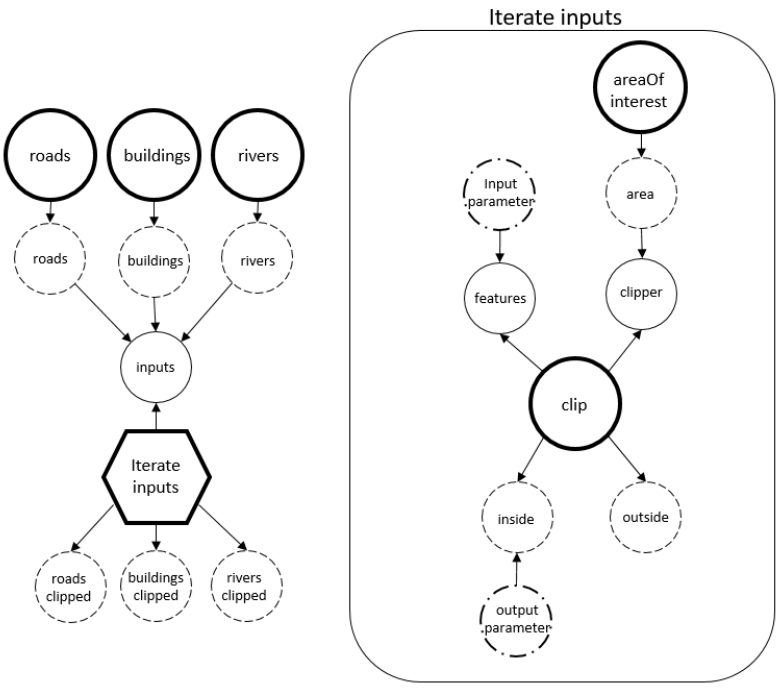


Figure 3.8: Example of Iterate Input

Remark. The execution for Iterate Inputs, Iterate Sets, and Iterate Multivalue can be performed in parallel, as the loop execution is independent of each other.

3.5 SUMMARY

In this chapter, we discuss which forms of composition can be recognized, and for each form how far can be decided on its validity. We define five levels of composability: Static Syntactic Composability, Dynamic Syntactic Composability, Structural Composability, Semantic Composability, and Qualitative Composability. The latter we do not discuss in this thesis.

Also, we discuss Structural Composability, which abstracts the composition as a directed acyclic graph and defines the rules for assessing composition well-formedness. We formalized the following constraints for a composition graph:

- The composition cannot contain dependency cycles.
- The inputs and outputs associated with one service need to contain different tags.
- The graph needs to be weakly connected.
- The composition cannot contain multi-edges.
- The composition cannot contain self-connections.
- The well-formedness of a composition with one or more conditional nodes can only be evaluated after the graph is divided into the possible scenarios.
- Cyclic-referencing is not allowed in subgraphs.
- Loop needs to be ensured to halt at compile-time.

Chapter 4 discusses the second level of composability, the Static Syntactic Composability.

Chapter 4
Static Syntactic Composability

4.1 INTRODUCTION

In Chapter 3, we defined five levels of composability, each of which verifies the composition according to some criteria, and we discussed the first level, Structural Composability. In this chapter, we address *Static Syntactic Composability*, i.e., the second level, which looks at geospatial web services and service composition as typed functions. In such functions, spatial data types play a fundamental role to characterize inputs and outputs, and consequently service types. We present here a type checking algorithm to prevent a class of execution errors based only on static information about the inputs and outputs provided by the service.

Section 4.2 introduces the type system $F_{S<:}$, capable of expressing types for web services. Section 4.3 defines the constructed types and abstract data types valid in $F_{S<:}$. Section 4.4 defines subtyping rules, the main mechanism to verify whether an input is valid, type-theoretically. Section 4.5 discusses type propagation operations, which allow to express dependencies of output types on input types. This is desired since classical static type systems do not allow to express type propagation, and thus, some form of parametric typing is needed. Lastly, Section 4.6 addresses the special cases of conditional, subgraph, and loop.

4.2 TYPE SYSTEM

Static Syntactic Composability is a service composition characteristic. It states that the output of a component service can only become the input of the next service if their data types are identical or the output data can be automatically coerced to the data type of the input parameter. We desire to verify this characteristic at compile-time with the goal of preventing a class of execution errors. An *execution error* is defined as an incorrect step that causes a program to behave in an unintended or unanticipated manner.

In Static Syntactic Composability, we intend to prevent run-time type errors by performing compile-time type-checking. I.e. we develop an algorithm that compares type signatures of outputs and inputs, and assesses whether the interaction between them is valid. In this thesis we define *type error* as an execution error caused by performing an operation on a value that is not of the appropriate type. We define a composition as *well-behaved* if it does not produce type errors at run-time. A well-behaved composition still can present execution errors. However, it is expected that type-checking prevents a significant amount of them.

The focus of this thesis is on compile-time verification, which cannot guarantee the absence of errors. It prevents the composition (and thus, execution) of processes that generate incorrect results or no results at all. This, in turn, may lead to more robust results, saving users time, removing unnecessary processing and data transfer over the Web.

We can use type systems to prevent errors during the execution of a program. A *Type System* is a collection of rules that label expressions with data type information, and that allow the inference of data type information on outcomes that a program generates. The rules state conditions under

which a program will not display a certain class of execution error. In this thesis, we introduce the type system $F_{S<:}$, designed to follow Cardelli's three expected properties of a type system:

- *decidably verifiable*: there is a type-checking algorithm capable of ensuring that a composition is well-behaved;
- *transparent*: it can predict whether the composition will type-check correctly, and if it fails, the reason is evident;
- *enforceable*: type declarations are preferentially checked at compile-time, and when that is not possible be checked at run-time.

In this thesis, the Cardelli formalism [71] is used to describe the type system. It applies three basic concepts:

- *expression*: a syntactic construct that expresses a value or value computation. An expression may contain free variables. Examples are: 1, $x + 1$, $true$, $x + y$.
- *data type*: static knowledge about the range of values that an expression can assume during the execution of a program. For example, an expression of type `boolean` can assume the values *true* and *false*, while an expression of type `integer` can assume the values $0, 1, \ldots$.
- *environment*: usually denoted by Γ, is a set of variables and their associated types during the execution of a particular program fragment. It is of the form $\{ x_1 : \tau_1, \ldots, x_n : \tau_n \}$ where x_k is a declared variable and τ_k is its type. The set of declared variables of an environment is denoted by $dom(\Gamma)$. The trivial environment is the empty environment, denoted by $\emptyset$.

The goal is to guide the implementation of a static type checker for geospatial web service compositions. We consider a environment, expression, or type to be *well-formed* if it is *properly constructed according to formal rules* [71]. Table 4.1 defines the basics judgments, formal assertions that relate environments, expressions, and types of the type system [71].

Table 4.1: Judgments for $F_{S<:}$

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash \tau$	τ is a well-formed type in Γ
$\Gamma \vdash e : \tau$	*e* is a well-formed expression of type τ in Γ. The ":" is seen as a has-type relation between expressions and types.

The last notation that we need to introduce is the Type Inference Rule. *Type Inference Rule* allows one to assert the validity of a judgment based on premise judgments that are known to be valid. In the notation, above the horizontal line are declared zero or more premise judgments $\Gamma_k \vdash \tau_k$, and below the line, at least, one conclusion judgment $\Gamma \vdash \tau$, that is valid if all the premise judgments are valid. Also, in each rule, one can specify a rule name. Equation 4.1 presents the type inference rule notation.

$$\frac{\begin{array}{c}\textsc{Rule Name}\\ \Gamma_1 \vdash \tau_1 \qquad \ldots \qquad \Gamma_n \vdash \tau_n\end{array}}{\Gamma \vdash \tau} \tag{4.1}$$

Table 4.2 presents the basic rules of the type system. The rule *Environment* $\emptyset$ states that the empty environment is well-formed. The rule *Environment Extension* allows one to extend an

environment by providing a variable, not in the domain of the environment and a well-formed type. The rule *Valid Variables* allows one to access variables from a well-formed environment. The rule *Type Construction* allows defining the valid types in the type system. Section 4.3 describes the types contained in the *Basic* set.

Table 4.2: Basic rules

ENVIRONMENT $\emptyset$

$$\frac{}{\emptyset \vdash \diamond}$$

ENVIRONMENT EXTENSION

$$\frac{\Gamma \vdash \tau \qquad x \notin dom(\Gamma)}{\Gamma, x : \tau \vdash \diamond}$$

VALID VARIABLES

$$\frac{\Gamma, x : \tau \vdash \diamond}{\Gamma, x : \tau \vdash x : \tau}$$

TYPE CONSTRUCTION

$$\frac{\Gamma \vdash \diamond \qquad K \in Basic}{\Gamma \vdash K}$$

4.3 TYPE DEFINITION

The proposed type system $F_{S<:}$ is an extension of Cardelli's type system $F_{1<:}$ [71]. Cardelli defined five basic types of interest to this thesis: `unit`, `string`, `integer`, `real`, `boolean`. In addition to the types described by Cardelli, we define a number of abstract types to express spatio-temporal concepts. Based on the OGC Simple Feature Access standard [57] and ISO 19125 [72] we identified the basic geometric types and their relations. Table 4.3 describes the geometry types covered in this thesis.

Table 4.3: Geometry types

Type	Description
`point`	A 0-dimensional geometric object that represents a position.
`linestring`	A 1-dimensional geometric object that represents a piecewise linear curve.
`polygon`	A 2-dimensional planar geometric object defined by one exterior boundary and zero or more interior boundaries. The boundaries are defined by closed and simple `linestrings`.
`geometrycollection`	A geometric object that is a collection of geometric objects.
`multipoint`	A `geometrycollection` of which all elements are 0-dimensional.
`multilinestring`	A `geometrycollection` of which all elements are LineStrings.
`multipolygon`	A `geometrycollection` of which all elements are Polygons.
`geometry`	Defined as a non-instantiable root class of the geometry hierarchy.

To overcome representation inconsistencies of bounding boxes as polygons, we add to the type system another abstract data type not covered by the Simple Feature Access Standard, namely the `bbox` type. It represents an axis-aligned bounding box which is guaranteed to contain one or more geometries, and which allows the reasoning on the geometries' coordinates range. Since the bounding box can also be represented by only the bottom left and top right corner coordinates, it must be conceptually distinguished from the Polygon type.

The definition of our temporal types is based on the OGC discussion paper A Conceptual Model and Text-based Notation for Temporal Geometry [73], in ISO 8601 [74], ISO 19108 [75], and WMS Time, a profile of ISO 8601 specifically for use in WMS. Table 4.4 describes the seven defined temporal types.

Table 4.4: Temporal types

Type	Description
`instant`	An instant in time with associated precision that represents all time points within the precision of the statement.
`period`	All time instants between two defined limiting instants. It is equivalent to TimeInterval in ISO 8601.
`multiinstant`	A set of Instants, either individually described or represented as `regularmultiinstant`.
`regularmultiinstant`	Sequence of consecutive Instants in constant time intervals.
`multiperiod`	A set of Periods, either individually described or represented as `regularmultiperiod`.
`regularmultiperiod`	Sequence of consecutive Periods with the same length of time in constant or variable time intervals. It is equivalent to Recurring-TimeInterval in ISO 8601.
`duration`	Length of time elapsed between two Instants. It is equivalent to TM_Duration in ISO 19108.

Remark. It is important to notice the difference between a temporal type and a time format. Each of the seven introduced types can have values in different formats. For example, all the following values belong to the type `instant`, however, they are portrayed in different formats (and precisions): 19850412, 1985-04-12, 1985-102, 1985-W15-5, 1985-W15, 1985-04, 1985. Syntactic Static Composability does not take into account the possible formats of a variable. This is addressed in Chapter 5, Dynamic Syntactic Composability, also with time zones and temporal coordinate systems.

The last series of abstract data types that we introduce allows the representation of coverages. A *coverage* is a function that associates a spatio-temporal domain with a range of values with defined types [76]. The definition of the coverage types was based on OGC GML Application Schema - Coverages [77] and ISO 19123:2005 [78]. Table 4.5 describes the seven defined coverage types.

Among the types defined by Cardelli, three categories are essential for the type system: *Record types*, *Set types*, and *Function types*. The first two are used to express the notion of feature types and feature collections. The Function type is used to define the Service type. Array and Variant types are also part of the type system $F_{S<:}$, however, for the sake of brevity they are not discussed in detail here. One can find more information about them in [71].

Definition 4.1 (Record type)**.** A record is a fixed size data structure that associates values with attributes. A *record type* prescribes which are the attribute names, and which is the data type associated with each of those names. A record type can contain zero or more attributes, and the attribute names must be distinct. The rule *Attribute Selection* is a value-level operation that allows the extraction of attribute value by name. Table 4.6 define rules for the Record type.

The Record type is used to express feature types.[1] ISO 19101 [79] defines a *feature* as an *abstraction of real world phenomena having common characteristics*. A *feature type* is a description of a

[1]This is an unfortunate clash of names. A Feature type does not relate to a type of a type system, but to a widely used concept in the geospatial sciences.

Table 4.5: Coverage types

Type	Description
`rectifiedgridcoverage`	A discrete point coverage in which the domain is a regular, equispaced grid of geometrically referenced points.
`gridcoverage`	A discrete point coverage in which the domain is a regular, equispaced grid, however, not geometrically referenced such as in RectifiedGridCoverage.
`referenceablegridcoverage`	A discrete point coverage in which the domain is a grid not equispaced.
`multipointcoverage`	A discrete coverage in which the domain is a `multipoint` geometry and maps each point to a value in the range set. A example of a value of type `multipointcoverage` is a point cloud.
`multicurvecoverage`	A discrete coverage in which the domain is a `multilinestring` geometry and maps each element to a value in the range set. A example of a value of type `multicurvecoverage` is a trajectory.
`multisurfacecoverage`	A discrete coverage in which the domain is a `multipolygon` geometry and maps each element to a value in the range set. Examples of values of type `multisurfacecoverage` are Thiessen polygons and TIN.

Table 4.6: Record type

$$\textsc{Record Type}\quad \frac{\Gamma \vdash \tau_1 \quad \dots \quad \Gamma \vdash \tau_n}{\Gamma \vdash \langle a_1 : \tau_1, \dots, a_n : \tau_n \rangle} \qquad \textsc{Record Value}\quad \frac{\Gamma \vdash v_1 : \tau_1 \quad \dots \quad \Gamma \vdash v_n : \tau_n}{\Gamma \vdash \langle a_1 \leftarrow v_1, \dots, a_n \leftarrow v_n \rangle : \langle a_1 : \tau_1, \dots, a_n : \tau_n \rangle}$$

$$\textsc{Attribute Selection}\quad \frac{\Gamma \vdash M : \langle a_1 : \tau_1, \dots, a_n : \tau_n \rangle}{\Gamma \vdash M.a_j : \tau_j}$$

group of features, in which one associates an attribute with a name and a type. These similarities in definition make the record types ideal to express feature types. For example, a feature type that represents a river can have the attributes *length* of type `real`, *name* of type `string`, and *geom* of type `linestring`. The type of the described feature is ⟨name : `string`, length : `real`, geom : `linestring`⟩. A *Feature instance* is defined as an individual of a specific feature type having assigned values to the feature attributes [79], in this way we can see it as a record value. To specify *Feature collections*, an arbitrary finite unordered collection of feature instances with the same feature type, we need Set types.

Definition 4.2 (Set type)**.** A set is an unordered finite collection of elements of the same data type. A *Set type* prescribes which is this common data type of all elements of the set. This common type is called as *inner type*.

$$\textsc{Set Type}\quad \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathbb{P}\, \tau} \tag{4.2}$$

The Set type is particularly useful when combined with Record types, in which a set of records is used to express feature collections. In addition to the defined types in [71], we defined one more type category in this thesis: Unions. One can see this type as an unlabeled Variant type.

Definition 4.3 (Union type)**.** A *Union type* is an unordered collection of types of the possible types of an expression. It is used when one cannot determine a type at compile-time, but one has knowledge that it belongs to a finite collection of possible types. The possible types are called *element types*. Provided the rule Union Value, and based on the properties of the *OR* operator, one can derive the rules Flatten Union, Simplify Union, and the Reduce Union. Table 4.7 define rules for the Union type.

Table 4.7: Union type

TYPE UNION

$$\frac{\Gamma \vdash A_1 \quad \ldots \quad \Gamma \vdash A_n}{\Gamma \vdash \text{union}(\ A_1, \ldots,\ A_n\ )}$$

UNION VALUE

$$\frac{\Gamma \vdash M : \text{union}(\ A_1, \ldots,\ A_n\ )}{\Gamma \vdash M : A_1\ \vee \ldots\ \vee\ \Gamma \vdash M : A_n}$$

FLATTEN UNION

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \Gamma \vdash C}{\Gamma \vdash \text{union}(\ A,\ \text{union}(\ B,\ C\ )\ ) = \text{union}(\ A,\ B,\ C\ )}$$

SIMPLIFY UNION

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash \text{union}(\ A,\ A,\ B\ ) = \text{union}(\ A,\ B\ )}$$

REDUCE UNION

$$\frac{\Gamma \vdash A}{\Gamma \vdash \text{union}(\ A\ ) = A}$$

To define service types, first we need to define the inference rules for function types. Table 4.8 defines those rules.

Table 4.8: Function type

TYPE FUNCTION

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

VALUE APPLICATION

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash Mv : B}$$

In the type system $F_{S<:}$ the central type is the *service type*. This type is shorthand notation for a special case of the Function type, in which the domain and range of the function are characterized by a Record type.

Definition 4.4 (Service type)**.** The *service type* is a shorthand notation to express functions with multiple arguments and results. The Record type is used to identify zero or more uniquely named typed arguments, and one or more uniquely named typed results. The named arguments are called *input identifiers*, and the named results *output identifiers*. The rule *Service Execution* assign a series of values to the input identifier. If the value has the same type of the input identifier, it is called a *valid input*. When all the assigned values are valid inputs the service execution is valid in the Static Syntactic Composability context. Table 4.9 specifies the rules for the Service type.

Table 4.9: Service type

SERVICE TYPE

$$\frac{\Gamma \vdash A_1 \quad \ldots \quad \Gamma \vdash A_n \quad \Gamma \vdash B_1 \quad \ldots \quad \Gamma \vdash B_m}{\Gamma \vdash \langle i_1 : A_1, \ldots, i_n : A_n \rangle \to \langle o_1 : B_1, \ldots, o_m : B_m \rangle}$$

SERVICE EXECUTION

$$\frac{\Gamma \vdash M : \langle i_1 : A_1, \ldots, i_n : A_n \rangle \to \langle o_1 : B_1, \ldots, o_m : B_m \rangle \quad \Gamma \vdash v_1 : A_1 \quad \ldots \quad \Gamma \vdash v_n : A_n}{\Gamma \vdash M\langle i_1 \leftarrow v_1, \ldots, i_n \leftarrow v_n \rangle : \langle o_1 : B_1, \ldots, o_m : B_m \rangle}$$

Expressions of service type can be related to a process called input realization. *Input realization* is a relation between two expressions of service type, where an output identifier of the first service is paired with an input identifier of the second service. The type of the output identifier in an input realization is called *realized type*. An input realization is valid if all possible values for the first service output identifier are valid inputs for the connected input identifier of the second service. The act of creating input realization between two or more services is called a *service composition*. When all input realizations in a service composition are valid, the services are considered to be Static Syntactic Composable.

In the next section, we extend the type system with the notion of subtyping.

4.4 SUBTYPING

Subtyping is a binary relation between types, which formalizes substitutability of one for the other, expressing that an element of a type is also an element of its supertype [71]. The notation $A <: B$ expresses that the type A is a subtype of B, meaning that for all values v of type A, v is automatically of type B. Therefore, we add judgments of the form $\Gamma \vdash A <: B$ to the type system, in cases where this makes sense. Table 4.10 presents the basic rules for subtyping.

Table 4.10: Basic rules for subtyping in $F_{S<:}$

REFLEXIVITY

$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: A}$$

TRANSITIVITY

$$\frac{\Gamma \vdash A <: B \quad \Gamma \vdash B <: C}{\Gamma \vdash A <: C}$$

SUBSUMPTION

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B}{\Gamma \vdash a : B}$$

If S is an expression of service type of an input identifier of type A, and a value $v : B$ with $B <: A$, then the subsumption relation guarantee that v is a valid input. This mechanism is the basis for the verification of Static Syntactic Composability. A series of services is considered static syntactic composed if all input identifier types are supertypes of the correspondent output identifier type.

The type *Top*, defined as the supertype of all types, is also added to the type system. This means that all well-typed expressions have type *Top*. Table 4.11 presents the rules for it.

We also add ad hoc subtyping rules for abstract data types. OGC [57] defines the following subtype relations for geometric types:

Table 4.11: Rules for type Top

TYPE TOP	SUB TOP
$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Top}}$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A <: \mathit{Top}}$

$$\texttt{multipoint} <: \texttt{geometrycollection}$$
$$\texttt{multilinestring} <: \texttt{geometrycollection}$$
$$\texttt{multipolygon} <: \texttt{geometrycollection}$$
$$\texttt{geometrycollection} <: \texttt{geometry}$$
$$\texttt{point} <: \texttt{geometry}$$
$$\texttt{linestring} <: \texttt{geometry}$$
$$\texttt{polygon} <: \texttt{geometry}$$

For the Coverage types, we define the following subtype relations. Two additional abstract types are also defined in [77], `coverage` and `discretecoverage`, which are based on non-instantiable classes.

$$\texttt{discretecoverage} <: \texttt{coverage}$$
$$\texttt{gridcoverage} <: \texttt{coverage}$$
$$\texttt{rectifiedgridcoverage} <: \texttt{coverage}$$
$$\texttt{referenceablegridcoverage} <: \texttt{coverage}$$
$$\texttt{multisurfacecoverage} <: \texttt{discretecoverage}$$
$$\texttt{multicurvecoverage} <: \texttt{discretecoverage}$$
$$\texttt{multipointcoverage} <: \texttt{discretecoverage}$$

OGC [73] defines the following subtype relations for temporal types:

$$\texttt{regularmultiinstant} <: \texttt{multiinstant}$$
$$\texttt{regularmultiperiod} <: \texttt{multiperiod}$$

Table 4.12 presents the subtyping relations for Record types, as defined in [71].

Table 4.12: Subtype rules for Record type

WIDTH SUBTYPE

$$\frac{\Gamma \vdash \tau_1 \quad \ldots \quad \Gamma \vdash \tau_{n+m}}{\Gamma \vdash \langle a_1 : \tau_1, \ldots, a_{n+m} : \tau_{n+m} \rangle <: \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle}$$

DEPTH SUBTYPE

$$\frac{\Gamma \vdash \tau_1 <: \rho_1 \quad \ldots \quad \Gamma \vdash \tau_n <: \rho_n}{\Gamma \vdash \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle <: \langle a_1 : \rho_1, \ldots, a_n : \rho_n \rangle}$$

We also define subtype rules for the Set and Union types. For Set types, the subtype relation depends in the subtype relation of its inner type. For Union types, we define two basic rules: *Union Unpack*, and *Union Collapse*. The first allows an element-wise verification of subtype relation in Union types while the second allows one to replace the entire Union type in the case that all its elements are a subtype of a common type. By combining the Union Unpack rule and Union Collapse rule, one can derive the *Union Subtype* rule. Table 4.13 is presented the additional subtyping rules.

Table 4.13: Subtype rules for Set and Union types

SET SUBTYPE

$$\frac{\Gamma \vdash A <: B}{\Gamma \vdash \mathbb{P}\, A <: \mathbb{P}\, B}$$

UNION UNPACK

$$\frac{\Gamma \vdash \text{union}(\, \tau_1, \ldots, \tau_n \,) \qquad \Gamma \vdash \tau <: \tau_k}{\Gamma \vdash \tau <: \text{union}(\, \tau_1, \ldots, \tau_n \,)}$$

UNION COLLAPSE

$$\frac{\Gamma \vdash \tau_1 <: \tau \qquad \ldots \qquad \Gamma \vdash \tau_n <: \tau}{\Gamma \vdash \text{union}(\, \tau_1, \ldots, \tau_n \,) <: \tau}$$

UNION SUBTYPE

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 \quad \ldots \quad \Gamma \vdash \tau_n \quad \Gamma \vdash \rho_1 \quad \ldots \quad \Gamma \vdash \rho_m \\ \forall\, k \in \{1, \ldots, n\} \,.\, \exists\, j \in \{1, \ldots, m\} \,.\, \Gamma \vdash \tau_k <: \rho_j\end{array}}{\Gamma \vdash \text{union}(\, \tau_1, \ldots, \tau_n \,) <: \text{union}(\, \rho_1, \ldots, \rho_m \,)}$$

Example 4.1. Consider a composition of three services, the first is *neighborhoods*, with no inputs and one output, *nbhd*, that returns polygon features that represent neighborhoods with their respective names.

The second is *accidents*, with no inputs and one output, *acdt*, that returns features with geometry of type `point` that represents traffic accidents, with the attributes *acdtTime*, time at which the accident occurred, and *acdtCat*, the category of the accident.

The third is *aggregate*, with two inputs: *ref*, that receives a feature collection with geometry of type `polygon` or `multipolygon`, and *pnt* a feature collection with geometry of type `point`. The service has one output, *agg*, that returns a feature collection with Polygon geometry, with an attribute *pntCount*, the number of points inside each polygon. We can represent their types in the following way:

$$\text{neighborhoods} : \langle\,\rangle \to \langle\, \text{nbhd} : \mathbb{P}\langle \text{geom} : \texttt{polygon},\ \text{name} : \texttt{string}\rangle\,\rangle$$

$$\text{accidents} : \langle\,\rangle \to \langle\, \text{acdt} : \mathbb{P}\langle \text{geom} : \texttt{point},\ \text{acdtTime} : \texttt{instant},\ \text{acdtCat} : \texttt{string}\rangle\,\rangle$$

$$\begin{aligned}\text{aggregate} : \langle\, \text{ref} : \mathbb{P}\langle \text{geom} : \text{union}(\, \texttt{polygon}, \texttt{multipolygon}\,)\rangle,\ \text{pnt} : \mathbb{P}\langle \text{geom} : \texttt{point}\rangle\,\rangle \\ \to \langle\, \text{agg} : \mathbb{P}\langle \text{geom} : \text{union}(\, \texttt{polygon}, \texttt{multipolygon}\,)\rangle,\ \text{pntCount} : \texttt{integer}\rangle\,\rangle\end{aligned}$$

Consider the composition as in Figure 4.1.

For this set of services to be static syntactic composed it is necessary that the following two subtype relations hold:

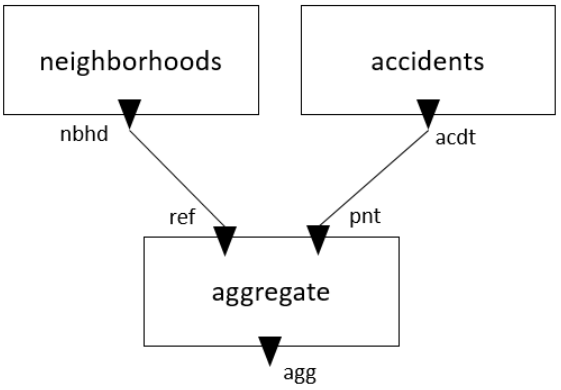


Figure 4.1: Example of composition

$$\mathbb{P}\langle \text{geom} : \texttt{polygon},\ \text{name} : \texttt{string}\rangle <: \mathbb{P}\langle \text{geom} : \text{union}(\ \texttt{polygon}, \texttt{multipolygon}\ )\rangle$$
$$\mathbb{P}\langle \text{geom} : \texttt{point},\ \text{acdtTime} : \texttt{instant},\ \text{acdtCat} : \texttt{string}\rangle <: \mathbb{P}\langle \text{geom} : \texttt{point}\rangle$$

For the first subtype relation, it can be derived by applying the Transitivity rule with the results of the following chain of two rules:

$$\textsc{Set Subtype}\ \frac{\textsc{Width Subtype}\ \dfrac{\Gamma \vdash \texttt{polygon} \qquad \Gamma \vdash \texttt{string}}{\Gamma \vdash \langle \text{geom} : \texttt{polygon},\ \text{name} : \texttt{string}\rangle <: \langle \text{geom} : \texttt{polygon}\rangle}}{\Gamma \vdash \mathbb{P}\langle \text{geom} : \texttt{polygon},\ \text{name} : \texttt{string}\rangle <: \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle}$$

$$\textsc{Set Subtype}\ \frac{\textsc{Depth Subtype}\ \dfrac{\textsc{Union Unpack}\ \dfrac{\textsc{Reflexivity}\ \dfrac{\Gamma \vdash \texttt{polygon}}{\Gamma \vdash \texttt{polygon} <: \texttt{polygon}}}{\Gamma \vdash \texttt{polygon} <: \text{union}(\ \texttt{polygon}, \texttt{multipolygon}\ )}}{\Gamma \vdash \langle \text{geom} : \texttt{polygon}\rangle <: \langle \text{geom} : \text{union}(\ \texttt{polygon}, \texttt{multipolygon}\ )\rangle}}{\Gamma \vdash \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle <: \mathbb{P}\langle \text{geom} : \text{union}(\ \texttt{polygon}, \texttt{multipolygon}\ )\rangle}$$

The second subtype relation follows in an analogous proof.

⌟

Remark. One can notice that the type checking was possible because input and output parameters used the same naming convention. In most of the cases, this will not happen, being necessary a mechanism for attribute renaming. If performed manually, this alignment of attribute names will be time-consuming and error prone. Chapter 5 introduces *attribute normalization*, a method that rename attributes based on semantic information added to attributes, which allow identifying attributes with different names that represent the same concept. Attribute normalization is a prerequisite for a correct verification of Static Syntactic Composability.

Finally, we define the subtype rules for Function and Service types. We define variance as the relation between subtyping in constructed types (set, array, functions) and subtyping between their type components. When the subtype relation is maintained in the same direction it is called a *covariant*, when the direction is inverted is called a *contravariant*. As described by Cardelli, function types are contravariant for arguments and covariant for results. The reasoning is that a

value $v : A'$ for any subtype A' of A is also a valid argument by the subsumption relation, and since it returns results with type B, it then obviously returns a value that also has any supertype B' of B. Therefore, a function of type $A \to B$ is also of a more general type $A' \to B'$. By combining the subtype rules for function type and record type, we can derive the rules for the service type. Table 4.14 presents the subtype rules for function type and service type. Both service type rules are derived from function subtype and width and depth subtype.

Table 4.14: Subtype rule for Service type

FUNCTION SUBTYPE

$$\frac{\Gamma \vdash A' <: A \qquad \Gamma \vdash B <: B'}{\Gamma \vdash A \to B <: A' \to B'}$$

SERVICE SUBTYPE

$$\frac{\Gamma \vdash A'_1 <: A_1 \quad \dots \quad \Gamma \vdash A'_n <: A_n \quad \Gamma \vdash B_1 <: B'_1 \quad \dots \quad \Gamma \vdash B_m <: B'_m}{\Gamma \vdash \langle i_1 : A_1, \dots, i_n : A_n \rangle \to \langle o_1 : B_1, \dots, o_m : B_m \rangle <: \langle i_1 : A'_1, \dots, i_n : A'_n \rangle \to \langle o_1 : B'_1, \dots, o_m : B'_m \rangle}$$

ARGUMENTS SUBTYPE

$$\frac{\Gamma \vdash A_1 \quad \dots \quad \Gamma \vdash A_{n+r} \quad \Gamma \vdash B_1 \quad \dots \quad \Gamma \vdash B_{m+s}}{\Gamma \vdash \langle i_1 : A_1, \dots, i_n : A_n \rangle \to \langle o_1 : B_1, \dots, o_{m+s} : B_{m+s} \rangle <: \langle i_1 : A_1, \dots, i_{n+r} : A_{n+r} \rangle \to \langle o_1 : B_1, \dots, o_m : B_m \rangle}$$

The notion of Service subtype is particularly important when one tries to substitute a particular service in a composition. As a prerequisite for substitutability, it is necessary that the substitute service's type has a subtype relation with the original service's type. This condition is naturally not sufficient to ensure that two services can be replaced, since other factors also take place as further discussed in Chapter 5, in Semantic Composability.

In the next section, we extend the type system with the notion of type propagation, which allows the definition of output identifier types based on the realized type of the input identifiers.

4.5 TYPE PROPAGATION

A common pattern in geospatial processing services is that the output type depends on the input types. So far, the presented type system is not able to do so. Type systems can be extended with the notion of *parametric polymorphism*, which introduces the notion of *type variable* that allow a data type to be expressed generically. This notion can be extended by constraining the possible types allowed in the type variable. This is called *bounded parametric polymorphism* [80].

In this section, we introduce a series of type operators to the type system, which allows a more detailed representation of the output types of services by providing a mechanism similar to bounded parametric polymorphism. Those operators are evaluated at compile-time and re-evaluated every time an input is realized. For example, suppose a function that performs a line simplification with two inputs: *ftr*, a feature collection with a geometry of type `linestring`; and *tolerance*, a number of type `real` which controls the degree of simplification. The output of the function is a feature collection of geometry type `linestring`.

$$\begin{aligned} \text{lineSimplification} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{linestring} \rangle,\ \text{tolerance} : \texttt{real}\ \rangle \\ \to \langle\ \text{simplified} : \mathbb{P}\langle \text{geom} : \texttt{linestring} \rangle\ \rangle \end{aligned}$$

Although this is a correct description of the function, it only provides a coarse description of the output. For example, suppose that for a valid input *ftr* is provided that has type $\mathbb{P}\langle$name :

$\texttt{string}, \text{length} : \texttt{real},\ \text{geom} : \texttt{linestring}\rangle$, the output type fails to describe that the output record also has the attributes *name* and *length*.

Due to the need to dynamically generate output type that reflects input provided, five type operators are defined in this section. They allow a more detailed representation of the output by means of type propagation of the input type into the output type. All operators are only valid as type declarations of output identifiers of service types.

Definition 4.5 (*typeof* Operator)**.** The *typeof* operator takes an input identifier within the same service and returns its type. The *typeof* operator can only be declared in type definitions of service outputs. Let S be a service with type $\langle i_1 : A_1, \ldots, i_n : A_n \rangle \to \langle \ldots \rangle$, we have that:

$$\mathbf{typeof}(\ i_k\ ) = A_k \tag{4.3}$$

In case that the input was realized the *typeof* operator returns the realized type. Let $v_k : B_k$ be a valid input, and consider the service composition $S\langle \ldots, i_k \leftarrow v_k, \ldots \rangle$.

$$\mathbf{typeof}(\ i_k\ ) = B \tag{4.4}$$

The operator can also be used to refer to types of record attributes, allowing attribute navigation paths. Consider the input identifier $i_k : \langle \ldots, a_j : B_j, \ldots \rangle$. We have:

$$\mathbf{typeof}(\ i_k.a_j\ ) = B_j \tag{4.5}$$

In an analogous way, one can also refer to the type of an attribute in the case of a feature collection.

Remark. The *typeof* operator extends the type theory in a similar way of parametric typing. However, it is restricted to types of the inputs of the attributes of the Service type.

Example 4.2. Consider the function *lineSimplification* with two inputs: *ftr*, a feature collection with geometry of type `linestring`; and *tolerance*, a number of type `real` which determines the degree of simplification of the geometry. This function returns that input feature collection but with the a simplified geometry. The *typeof* operator can express this propagation of type as follows:

$$\begin{aligned}\text{lineSimplification} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{linestring}\rangle,\ \text{tolerance} : \texttt{real}\ \rangle \\ \to \langle\ \text{simplified} : \mathbf{typeof}(\ \text{ftr}\ )\ \rangle\end{aligned}$$

Before any value is assigned to the inputs the *typeof* operator calculate its value based on the input type of the service. In this way *lineSimplification* has type:

$$\begin{aligned}\text{lineSimplification} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{linestring}\rangle,\ \text{tolerance} : \texttt{real}\ \rangle \\ \to \langle\ \text{simplified} : \mathbb{P}\langle \text{geom} : \texttt{linestring}\rangle\ \rangle\end{aligned}$$

Consider a valid input v_1 for *ftr* and a valid input v_2 for tolerance:

$$v_1 : \mathbb{P}\langle \text{name} : \texttt{string}, \text{length} : \texttt{real},\ \text{geom} : \texttt{linestring}\rangle$$

When this value is realized the type propagation re-evaluates and we deduce that:

$$\begin{aligned}\text{lineSimplification}\langle\ \text{ftr} \leftarrow v_1, \text{tolerance} \leftarrow v_2\ \rangle.\text{simplified} : \\ \mathbb{P}\langle \text{name} : \texttt{string},\ \text{length} : \texttt{real},\ \text{geom} : \texttt{linestring}\rangle\end{aligned}$$

⌟

Definition 4.6 (*unset* Operator)**.** The *unset* type operator take as input a Set type and returns its inner type.

$$\mathbf{unset}(\ \mathbb{P}\ \tau\ ) = \tau \tag{4.6}$$

Example 4.3. Consider the function *findNearest* with two inputs: *ftr*, a feature collection with geometry type `point`; and *ref*, a feature of geometry type `point`. The function returns the closest point in the set to the reference geometry, ensuring that only one point is returned. The type operator *unset* combined with the type operator *typeof* can express the type propagation:

$$\begin{aligned}\text{findNearest} : \langle\ \text{ftr} : \mathbb{P}\langle\text{geom} : \texttt{point}\rangle,\ \text{ref} : \langle\text{geom} : \texttt{point}\rangle\ \rangle \\ \rightarrow \langle\ \text{nearest} : \mathbf{unset}(\ \mathbf{typeof}(\ \text{ftr}\ )\ )\ \rangle\end{aligned}$$

Before any value is assigned to the inputs, *findNearest* has type:

$$\begin{aligned}\text{findNearest} : \langle\ \text{ftr} : \mathbb{P}\langle\text{geom} : \texttt{point}\rangle,\ \text{ref} : \langle\text{geom} : \texttt{point}\rangle\ \rangle \\ \rightarrow \langle\ \text{nearest} : \langle\text{geom} : \texttt{point}\rangle\ \rangle\end{aligned}$$

Consider a valid input v_1 for *ftr* and a valid input v_2 for *ref*:

$$v_1 : \mathbb{P}\langle\text{name} : \texttt{string},\ \text{address} : \texttt{string},\ \text{geom} : \texttt{point}\rangle$$

Applying the type propagation we deduce that:

$$\begin{aligned}\text{findNearest}\langle\ \text{ftr} \leftarrow v_1, \text{ref} \leftarrow v_2\ \rangle.\text{nearest} : \\ \langle\text{name} : \texttt{string},\ \text{address} : \texttt{string},\ \text{geom} : \texttt{point}\rangle\end{aligned}$$

⌟

Definition 4.7 (*addAttrs* Operator)**.** The *addAttrs* type operator, denoted by $\oplus$, takes two Record types and returns a Record type with the attributes merged:

$$\begin{aligned}\langle a_1 : \tau_1, \ldots,\ a_{n+m} : \tau_{n+m}\rangle \oplus \langle a_{n+1} : \rho_{n+1}, \ldots,\ a_{n+m+j} : \rho_{n+m+j}\rangle = \\ \langle a_1 :\ \tau_1, \ldots, a_n : \tau_n,\ a_{n+1} : \rho_{n+1}, \ldots,\ a_{n+m+j} : \rho_{n+m+j}\rangle\end{aligned} \tag{4.7}$$

In the case that the Record types have a common attribute the type is overwritten by the attribute in the record on the right-hand side. This allows one to update types in a record.

$$\langle a_1 : \tau_1, \ldots,\ a_n : \tau_n\rangle \oplus \langle a_1 : \rho\rangle = \langle a_1 : \rho, a_2 : \tau_2, \ldots,\ a_n : \tau_n\rangle \tag{4.8}$$

This implies that, if A and B are Record types with common attributes but different types: $A \oplus B \neq B \oplus A$. The *addAttrs* operator can also be applied to Set types in the case that its inner type is a Record type:

$$\begin{aligned}\mathbb{P}\langle a_1 : \tau_1, \ldots,\ a_{n+m} : \tau_{n+m}\rangle \oplus \langle a_{n+1} : \rho_{n+1}, \ldots,\ a_{n+m+j} : \rho_{n+m+j}\rangle = \\ \langle a_1 :\ \tau_1, \ldots, a_n : \tau_n,\ a_{n+1} : \rho_{n+1}, \ldots,\ a_{n+m+j} : \rho_{n+m+j}\rangle\end{aligned} \tag{4.9}$$

Example 4.4. Consider the function *distanceFromReference* with two inputs: *ftr*, a feature collection with geometry type `point`; and *ref*, a feature of geometry type `point`. The function returns that feature collection and adds an attribute *distance* of type `real`. The type operator *addAttrs* combined with the type operator *typeof* can express the type propagation:

$$\text{distanceFromReference} : \langle\ \text{ftr} : \mathbb{P}\langle\text{geom} : \texttt{point}\rangle,\ \text{ref} : \langle\text{geom} : \texttt{point}\rangle\ \rangle \\ \rightarrow \langle\ \text{ftrDist} : \mathbf{typeof}(\ \text{ftr}\ ) \oplus \langle\text{distance} : \texttt{real}\rangle\ \rangle$$

Before any value is assigned to the inputs, *distanceFromReference* has type:

$$\text{distanceFromReference} : \langle\ \text{ftr} : \mathbb{P}\langle\text{geom} : \texttt{point}\rangle,\ \text{ref} : \langle\text{geom} : \texttt{point}\rangle\ \rangle \\ \rightarrow \langle\ \text{ftrDist} : \mathbb{P}\langle\text{geom} : \texttt{point},\ \text{distance} : \texttt{real}\rangle\rangle\ \rangle$$

Consider a valid input v_1 for *ftr* and a valid input v_2 for *ref*:

$$v_1 : \mathbb{P}\langle\text{name} : \texttt{string},\ \text{address} : \texttt{string},\ \text{geom} : \texttt{point}\rangle$$

Applying the type propagation we deduce that:

$$\text{distanceFromReference}\langle\ \text{ftr} \leftarrow v_1, \text{ref} \leftarrow v_2\ \rangle.\text{ftrDist} : \\ \mathbb{P}\langle\text{name} : \texttt{string},\ \text{address} : \texttt{string},\ \text{geom} : \texttt{point},\ \text{distance} : \texttt{real}\rangle$$

⌟

Example 4.5. Consider the function *buffer* with two inputs: *ftr*, a feature collection with geometry type `geometry`; and *distance*, the buffer distance of type `real`. The function returns that feature collection with the geometry modified to `polygon` or `multipolygon`, which depends on the input geometry and the position of the features. The type operator *addAttrs* combined with the type operator *typeof* can express the type propagation:

$$\text{buffer} : \langle\ \text{ftr} : \mathbb{P}\langle\text{geom} : \texttt{geometry}\rangle,\ \text{distance} : \texttt{real}\ \rangle \\ \rightarrow \langle\ \text{buffered} : \mathbf{typeof}(\ \text{ftr}\ ) \oplus \langle\text{geom} : \text{union}(\ \texttt{polygon},\ \texttt{multipolygon}\ )\rangle\ \rangle$$

Consider a valid input v_1 for *ftr* and a valid input v_2 for distance:

$$v_1 : \mathbb{P}\langle\text{name} : \texttt{string},\ \text{length} : \texttt{real},\ \text{geom} : \texttt{multilinestring}\rangle$$

Applying the type propagation we deduce that:

$$\text{buffer}\langle\ \text{ftr} \leftarrow v_1, \text{distance} \leftarrow v_2\ \rangle.\text{buffered} : \\ \mathbb{P}\langle\text{name} : \texttt{string},\ \text{length} : \texttt{real},\ \text{geom} : \text{union}(\ \texttt{polygon},\ \texttt{multipolygon}\ )\rangle$$

⌟

Example 4.6. Consider the function *joinById* with two inputs: *ftr1* and *ftr2*, two feature collections with an identifier attribute. The function returns a set of Records that have the combination of the attributes of both underlying record types. In the case that an attribute with the same name exists in both features, the value in the result comes from *ftr2*. The type operator *addAttrs* combined with the type operator *typeof* can express the type propagation:

$$\text{joinById} : \langle\ \text{ftr1} : \mathbb{P}\langle\text{id} : \texttt{string}\rangle,\ \text{ftr2} : \mathbb{P}\langle\text{id} : \texttt{string}\rangle\ \rangle \\ \rightarrow \langle\ \text{joined} : \mathbf{typeof}(\ \text{ftr1}\ ) \oplus \mathbf{typeof}(\ \text{ftr2}\ )\ \rangle$$

Before any value is assigned to the inputs, *joinById* has type:

$$\text{joinById} : \langle\ \text{ftr1} : \mathbb{P}\langle \text{id} : \texttt{string}\rangle,\ \text{ftr2} : \mathbb{P}\langle \text{id} : \texttt{string}\rangle\ \rangle \\ \rightarrow \langle\ \text{joined} : \mathbb{P}\langle \text{id} : \texttt{string}\rangle\ \rangle$$

Consider a valid input v_1 for *ftr1*:

$$v_1 : \mathbb{P}\langle \text{id} : \texttt{string},\ \text{name} : \texttt{string},\ \text{geom} : \texttt{polygon}\rangle$$

And a valid input v_2 for *ftr2*:

$$v_2 : \mathbb{P}\langle \text{id} : \texttt{string},\ \text{name} : \texttt{string},\ \text{population} : \texttt{integer}\rangle$$

Applying the type propagation we deduce that:

$$\text{joinById}\langle\ \text{ftr1} \leftarrow v_1, \text{ftr2} \leftarrow v_2\ \rangle.\text{joined} : \\ \mathbb{P}\langle \text{id} : \texttt{string},\ \text{name} : \texttt{string},\ \text{geom} : \texttt{polygon},\ \text{population} : \texttt{integer}\rangle$$

⌟

Definition 4.8 (*valueof* Operator)**.** Let A be the set of input identifier names of a service type. The *valueof* operator receives as an input an element a of A, such that $\mathbf{typeof}(a) = \texttt{string}$ and returns the value assigned to this attribute. This operator can only be used in the context of a record type, and it is used to dynamically create attributes in a Record type.

Let S be service with type $\langle a : \texttt{string}, \dots\rangle \rightarrow \langle \dots\rangle$, v a value of type `string` and the service execution $S\langle a \leftarrow v, \dots\rangle$, we have that:

$$\mathbf{valueof}(\ a\ ) = v \tag{4.10}$$

This value should follow all the conditions applied to attribute names, such as not generate duplicate attribute names. In the case that the operator evaluates to an invalid attribute name a type error is raised. In the case of an input realization in which the value cannot be evaluated at compile-time the attribute in which the *valueof* operator is used is removed.

Example 4.7. Consider the function *Max* with four inputs: *ftr*, a feature collection with geometry type `polygon`; *pts*, a feature collection with geometry type `point`; *infield*, a `string` which represent the name of one attribute of the set of point features that will be used for the calculation of the maximum value; *outfield*, a `string` which represent the name of the attribute that will be added to the set of polygon features. The function returns the same set of polygons added a new attributed of type `real` and name provided in the input. The type operator *valueof* combined with the type operators *typeof* and *addAttrs* can express the type propagation:

$$\text{max} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle,\ \text{pts} : \mathbb{P}\langle \text{geom} : \texttt{point}\rangle, \\ \text{infield} : \texttt{string},\ \text{outfield} : \texttt{string}\ \rangle \\ \rightarrow \langle\ \text{maximum} : \mathbf{typeof}(\ \text{ftr}\ ) \oplus \langle \mathbf{valueof}(\ \text{outfield}\ ) : \texttt{real}\rangle\ \rangle$$

Consider a valid input v_1 for *ftr*:

$$v_1 : \mathbb{P}\langle \text{name} : \texttt{string},\ \text{geom} : \texttt{polygon}\rangle$$

The valid input v_2 for *pts*:

$$v_2 : \mathbb{P}\langle \text{height} : \texttt{real},\ \text{geom} : \texttt{point}\rangle$$

And the value v_3 = "height" for *infield*, and v_4 = "maxHeight" for *outfield*. Applying the propagation of type we deduce that:

$$\max\langle\ \text{ftr} \leftarrow v_1, \text{pts} \leftarrow v_2, \text{infield} \leftarrow v_3, \text{outfield} \leftarrow v_4\ \rangle.\text{maximum} :$$
$$\mathbb{P}\langle \text{name} : \texttt{string},\ \text{geom} : \texttt{polygon},\ \text{maxHeight} : \texttt{real}\rangle$$

Remark. Before the input value for *outfield* is known, the attribute of identifier *outfield* cannot be realized, meaning that the output type of maximum will be equivalent to:

$$\mathbf{typeof}(\ \text{ftr}\ ) \oplus \langle\rangle = \mathbf{typeof}(\ \text{ftr}\ )$$

⌟

Definition 4.9 (*remAttrs* Operator)**.** The *remAttrs* type operator, denoted by $\ominus$, receives as input two Record types and returns a Record type with the attributes which belongs to the left Record, but does not belong to the right one:

$$\langle v_1 : \tau_1, \ldots,\ v_{n+m} : \tau_{n+m}\rangle \ominus \langle v_{n+1} : \tau_{n+1}, \ldots,\ v_{n+m} : \tau_{n+m}\rangle \equiv \langle v_1 :\ \tau_1, \ldots,\ v_n : \tau_n\rangle \quad (4.11)$$

Trying to remove attributes that does not exists is valid:

$$\langle v_1 : \tau_1, \ldots,\ v_n : \tau_n\rangle \ominus \langle v_n : \tau_n,\ v_{n+1} : \tau_{n+1}\rangle \equiv \langle v_1 :\ \tau_1, \ldots,\ v_{n-1} : \tau_{n-1}\rangle \quad (4.12)$$

We can also apply the *remAttrs* operator to Set types in the case that its inner type is a Record type.

Remark. The types of the attributes in the records of the right side of *remAttrs* operator are not relevant to the calculation, which means that we can simplify the notation omitting those types.

$$\langle v_1 : \tau_1, \ldots,\ v_{n+m} : \tau_{n+m}\rangle \ominus \langle v_{n+1}, \ldots,\ v_{n+m}\rangle = \langle v_1 :\ \tau_1, \ldots,\ v_n : \tau_n\rangle \quad (4.13)$$

Example 4.8. Consider the function *removeAttribute* with two inputs: *ftr*, a feature collection; *name*, a `string` which represent the name of the attribute to be removed. The function returns the same feature collection as the inputs removed the desired attributed. The type operator *remAttrs* combined with the type operators *typeof* and *valueof* can express the type propagation:

$$\text{removeAttribute} : \langle\ \text{ftr} : \mathbb{P}\langle\rangle,\ \text{name} : \texttt{string}\ \rangle$$
$$\rightarrow \langle\ \text{attrRemoved} : \mathbf{typeof}(\ \text{ftr}\ ) \ominus \langle \mathbf{valueof}(\ \text{name}\ )\rangle\ \rangle$$

Consider a valid input v_1 for *ftr*:

$$v_1 : \mathbb{P}\langle \text{name} : \texttt{string},\ \text{population} : \texttt{integer},\ \text{geom} : \texttt{multipolygon}\rangle$$

And the value v_2 = "geom" for *name*. Applying the propagation of type we deduce that:

$$\text{removeAttribute}\langle\ \text{ftr} \leftarrow v_1, \text{name} \leftarrow v_2\ \rangle.\text{attrRemoved} :$$
$$\mathbb{P}\langle \text{name} : \texttt{string},\ \text{population} : \texttt{integer}\rangle$$

⌟

It is important to notice that verifying the composability based on specifications is overly restrictive, having false negatives. Consider the function *Dissolve*, that receives a set of polygons and performs the merge of all the polygons that intersects. The possible results are of type `polygon` or `multipolygon`, therefore, further compose with a function that requires a `polygon` as an input in a general case is not valid. However, it still exists inputs that all the polygons are dissolved into a single polygon making the functions composable, meaning that exists false negatives.

4.6 CONDITIONAL AND LOOP TYPE CHECKING

In Chapter 3, we introduced three categories of nodes that are special cases of the service node, namely, conditional, subgraph, and loop. The *Conditional* node allows a boolean evaluation of a condition at run-time. It leads to one of two outputs: *trueOut* if the condition evaluates to true, and *falseOut* otherwise. This allows exclusive branching the composition into two different scenarios, meaning that the composition will use only one of the output scenarios. In compile-time verification, the type checker cannot predict which branch of the composition will be used, thus necessitating the evaluation of both scenarios of the composition. We can characterize in our typing system the conditional node with a Service type in the following way:

$$\text{conditional} : \langle\, \text{input} : \tau \,\rangle \rightarrow \langle\, \text{trueOut} : \mathbf{typeof}(\, \text{input}\,),\ \text{falseOut} : \mathbf{typeof}(\, \text{input}\,) \,\rangle$$

This node just passes the input type to the output ports, without any modification to the data. We can apply this construct to decide at run-time between elements in a union type.

Example 4.9. Consider the services organized as in Figure 4.2 and the following type definitions for each service:

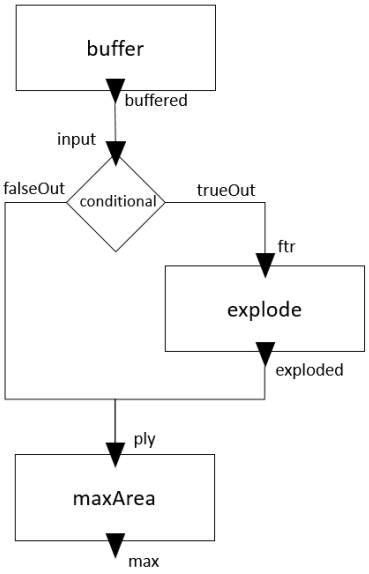


Figure 4.2: Example of composition with conditional

$$\text{buffer} : \langle\, \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{geometry}\rangle,\ \text{distance} : \texttt{real}\,\rangle \rightarrow \langle\, \text{buffered} : \langle \text{geom} : \text{union}(\,\texttt{polygon},\ \texttt{multipolygon}\,)\rangle\,\rangle$$

$$\text{explode} : \langle\, \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{multipolygon}\rangle\,\rangle \rightarrow \langle\, \text{exploded} : \mathbf{typeof}(\, \text{ftr}\,) \oplus \langle \text{geom} : \texttt{polygon}\rangle\,\rangle$$

$$\text{maxArea} : \langle\, \text{ply} : \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle\,\rangle \rightarrow \langle\, \text{max} : \mathbf{unset}(\, \mathbf{typeof}(\, \text{ply}\,)\,)\,\rangle$$

The *explode* process explodes multipolygons into polygons. The conditional structure is used to verify at run-time whether the buffer output is realized as `polygon` or `multipolygon`. This conditional requires the input to be a collection of features with geometry type `polygon` or `multipolygon`,

and in the trueOut the feature is returned in case of the geometry be `multipolygon`, and the falseOut is returned in case of `polygon`. We formalize the conditional as:

$$\begin{aligned}\text{conditional} : \langle\ \text{input} :& \mathbb{P}\langle \text{geom} : \text{union}(\ \texttt{polygon},\ \texttt{multipolygon}\ )\rangle\ \rangle \\ & \rightarrow \langle\ \text{trueOut} : \mathbf{typeof}(\ \mathit{input}\ ) \oplus \langle \text{geom} : \texttt{multipolygon}\rangle, \\ & \quad\ \text{falseOut} : \mathbf{typeof}(\ \text{input}\ ) \oplus \langle \text{geom} : \texttt{polygon}\rangle\ \rangle\end{aligned}$$

The type checker considers both output ports of the conditional when it verifies the composition.

⌟

Chapter 3 also introduces subgraph nodes, a service node that can be associated with parametric subgraphs. Parametric graphs are graphs with one or more input parameter node, and one or more output parameter node. To define types for subgraph nodes we first need to define the type for the input parameter node and output parameter node. Both nodes have no inherent type associated, their type is inferred from the types of the connected services. The input parameter can connect to one or more services, and it is required to have a type that the value handover to each service will be valid. We restrict that the input types of the connected services to be in a subtype relation, in this way the type that is the subtype of all others is a valid type for the input parameter. The type of the output parameter node is easily defined. Since it can only be connected to one service output, its type becomes the same as the service output.

Example 4.10. Consider the services organized as in Figure 4.3 and the following type definitions for each service:

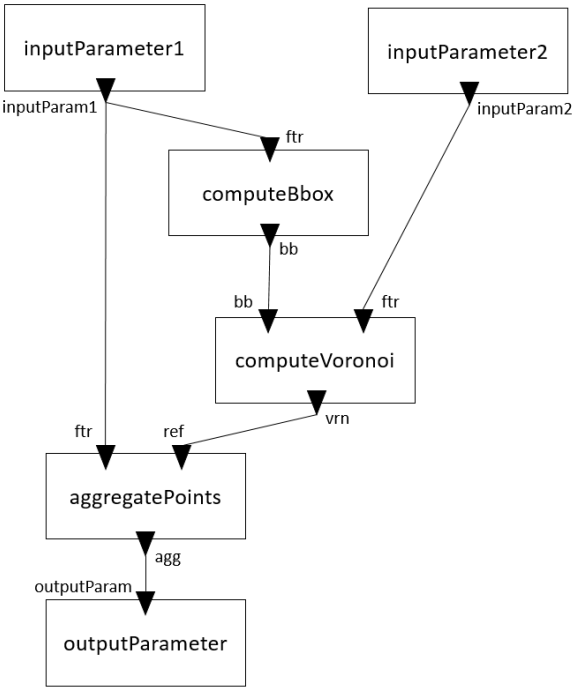


Figure 4.3: Example of subgraph type inference

$$\begin{aligned}\text{computeVoronoi} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{point}\rangle,\ \text{bb} : \texttt{bbox}\ \rangle \\ \rightarrow \langle\ \text{vrn} : \mathbf{typeof}(\ \text{ftr}\ ) \oplus \langle \text{geom} : \texttt{polygon}\rangle\ \rangle\end{aligned}$$

$$\text{computeBbox} : \langle \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{geometry}\rangle \rangle \rightarrow \langle\ \text{bb} : \texttt{bbox}\ \rangle$$

$$\begin{aligned}\text{aggregatePoints} : \langle \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{point}\rangle,\ \text{ref} : \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle \rangle \\ \rightarrow \langle\ \text{agg} : \mathbf{typeof}(\ \text{ref}\ ) \oplus \langle \text{count} : \texttt{integer}\rangle\ \rangle\end{aligned}$$

The $inputParam2$ type can be directly inferred since it is only connected with one input. Its type becomes $\mathbb{P}\langle \text{geom} : \texttt{point}\rangle$. Since $inputParam1$ is connected to two inputs, and as they are in a subtype relation $\mathbb{P}\langle \text{geom} : \texttt{point}\rangle <: \mathbb{P}\langle \text{geom} : \texttt{geometry}\rangle$, the type for $inputParam1$ is $\mathbb{P}\langle \text{geom} : \texttt{point}\rangle$.

The $outputParam$ type is also easily inferred, since it is only connected to one output. Its type becomes $\mathbf{typeof}(\ \text{ref}\ ) \oplus \langle \text{count} : \texttt{integer}\rangle$. Before any input is realized this type evaluates to $\mathbb{P}\langle \text{geom} : \texttt{polygon},\ \text{count} : \texttt{integer}\rangle$.

⌟

Chapter 3 also defines loop nodes. The loop nodes are also associated with parametric graphs, and the types definition behaves in a similar way as described for subgraph nodes. For each type of loop introduced in Chapter 3, we can define the types for input and output:

- Iterate Inputs: behaves in the same way as the subgraph node.
- Iterate Sets: types of inputs and outputs are set types with inner types the same as the defined types for input and output parameter nodes.
- Iterate Multivalue: the *value* input is a set type with inner type the same as the defined type for the input parameter node.

Example 4.11. Consider an Iterate Multivalue node associated with a subgraph that performs a buffer operation, which allows multiple executions of the buffer operation with different distance values for the same input. Figure 4.4 shows the graph representation of the Iterate Multivalue node and a subgraph.

$$\begin{aligned}\text{buffer} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{geometry}\rangle,\ \text{distance} : \texttt{real}\ \rangle \\ \rightarrow \langle\ \text{buffered} : \mathbf{typeof}(\ \text{ftr}\ ) \oplus \langle \text{geom} : \text{union}(\ \texttt{polygon},\ \texttt{multipolygon}\ )\rangle\ \rangle\end{aligned}$$

The types for the nodes in the subgraph are defined as follows. The node *value* receives the type `real`, the node *inputParam* is typed as $\mathbb{P}\langle \text{geom} : \texttt{geometry}\rangle$, and the node *outputParam* is typed as $\mathbf{typeof}(\ \text{ftr}\ ) \oplus \langle \text{geom} : \text{union}(\ \texttt{polygon},\ \texttt{multipolygon}\ )\rangle$. Before any input is realized this type is evaluated to $\mathbb{P}\langle \text{geom} : \text{union}(\ \texttt{polygon},\ \texttt{multipolygon}\ )\rangle$. In this way, we can define the type for the Iterate Multivalue node as:

$$\begin{aligned}\text{iterateMultivalue} : \langle\ \text{inputs} : \mathbb{P}\langle \text{geom} : \texttt{geometry}\rangle,\ \text{values} : \mathbb{P}(\ \texttt{real}\ )\ \rangle \\ \rightarrow \langle\ \text{buffered} : \mathbb{P}\langle \text{geom} : \text{union}(\ \texttt{polygon},\ \texttt{multipolygon}\ )\ \rangle\end{aligned}$$

⌟

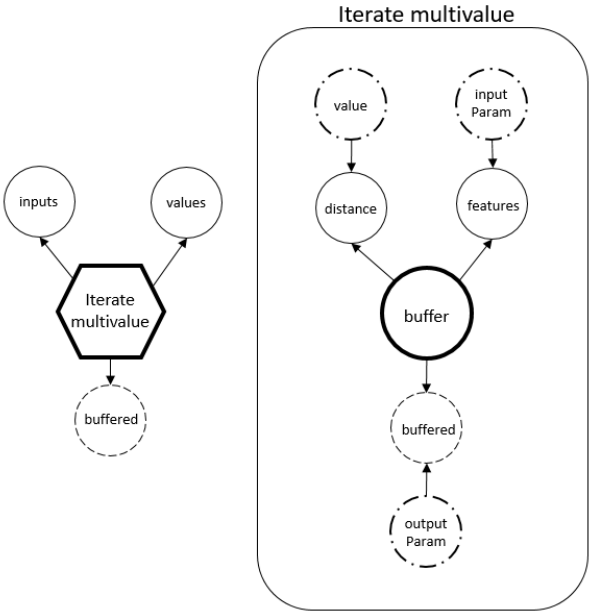


Figure 4.4: Example of loop type inference

4.7 SUMMARY

In this chapter, we discussed Static Syntactic Composability, in which we evaluated service compositions using the Cardelli type system $F_{1<:}$, augmented with an additional type, the Service type. This Service is a special case of the function type, which allows one to express a service as a function with zero or more named inputs and one or more named outputs. We also defined abstract data types, to express concepts about geometry, coverage, and time, and type propagation operators, which allows one to express the output types of services based on the input types. Lastly, we discussed how to type-check compositions with subgraph, conditional, and loop nodes.

Chapter 5 discusses Dynamic Syntactic Composability and Semantic Composability.

Chapter 5
Dynamic Syntactic and Semantic Composability

5.1 INTRODUCTION

In Chapter 3, we defined five levels of composability and discussed the first level, Structural Composability. In Chapter 4, we discussed Static Syntactic Composability, which introduces the Type System $F_{S<:}$, which allows the assignment of types to Web Services. In this chapter, we discuss two more levels of composability: Dynamic Syntactic Composability and Semantic Composability. We verify the first with preconditions and postconditions in Hoare Logic. This defines the most common expressions when dealing with geometry, coverage, and temporal types, and how the preconditions and postconditions of the composition can be derived based on the interactions between inputs and outputs. In Semantic Composability, we consider any service to be semantically enabled, and we discuss the semantics of input and output relation, the substitutability of services, the normalization of attribute names, and workflow semantics.

Section 5.2 extends Hoare Logic with the notion of input and output, which allows the verification of constraints which the type system proposed in Chapter 4 cannot express. Section 5.3 discusses in which forms one can define Semantic Composability, and proposes mechanisms for service substitution and name normalization of equivalent attributes consistently across the composition.

5.2 DYNAMIC SYNTACTIC COMPOSABILITY

Dynamic Syntactic Composability verifies input restrictions that the proposed type system of the previous chapter cannot address. This includes the use of service states, coordinate systems, time zones, interactions between inputs, restriction in the value range, and others. A way to express these restrictions is to use Hoare Logic preconditions. Section 5.2.1 introduces that formalism.

5.2.1 Hoare Logic

Hoare logic is a formal system for reasoning about the correctness of computer programs [19]. The verification is based on specifications: *preconditions*, which are conditions required for the program run correctly; and *postconditions*, which are conditions that are guaranteed to hold after a correct execution. The central concept in Hoare Logic is the *Hoare triple*, for which we use the following notation: $\{P\}\ C\ \{Q\}$. Here, P is the precondition, and Q is the postcondition, and both are described in predicate logic.[1] This notation allows the specification of an instruction C. The semantics of the relation is that *"If P holds before the execution of C, then Q will hold after the execution."* For an instruction C, or a batch of such instructions, the most interesting assertions are the weakest precondition and the strongest postcondition.

[1]The original notation is $P\{C\}Q$, a simple notation with the emphasis on the code C; however, the presented one is currently widely used and de-emphasizes code over conditions.

Definition 5.1 (Strongest postcondition)**.** If $\{P\}\ C\ \{Q\}$ and if for all Q' such that $\{P\}\ C\ \{Q'\}$, and $Q \Rightarrow Q'$, then Q is the *strongest postcondition* of C with respect to P.

Definition 5.2 (Weakest precondition)**.** If $\{P\}\ C\ \{Q\}$ and if for all P' such that $\{P'\}\ C\ \{Q\}$, and $P' \Rightarrow P$, then P is the *weakest precondition* of C with respect to Q.

In Dynamic Syntactic Composability, we abstractly view a web service as a function with zero or more arguments (inputs) and one or more results (outputs). We assume that only the precondition, postcondition, and service type are known while details of the service implementation are unknown. The goal of this section is to *describe inference rules to derive specifications for a composite service from the available specifications of the individual services*. This allows reasoning about the validity of a service composition.

Pre- and postconditions are expressions in Predicate logic. Possible variables in the expressions refer to inputs and outputs of the service as defined in the service type, or service states, or attributes of input or output records (accessed by attribute selection rule), or even metadata. *Metadata* consists of associations between a service state, input, output, or an attribute on one hand and a value that is maintained by the service on the other. This value is constant, and we consider it to be known by all services in the composition at compile-time. Such an association is denoted by $a{\uparrow}b$, where a may be a state, input, output, or attribute expression, and b is the metadata attribute (name). The allowed metadata attribute names are context-dependent on the type of the variable a. State variables are written with a starting uppercase letter while input and output with a lowercase letter. One can only differentiate an output from input by comparing with the service type signature. The state after the execution of a service operation is decorated with a dash, for example if the state before the execution is S, and then after execution is S'.

Besides variables, we allow the use of constants, functions, and predicates (here defined as boolean-valued functions). Table 5.1 presents the syntax for pre- and postconditions in Backus-Naur Form (BNF).

Example 5.1. Consider a function *getBuildings*, which receives two inputs, *crs*, a `string` representing a coordinate system, and a bounding box *bb*:`bbox`. The value for *crs* needs to be one of the possible CRS that the service can handle, i.e. belong to the set *availableCRS*. The value for *bb* needs to overlap the bounding box of all features, *layerBBOX*. The operation has as output a feature collection *bldg* of geometry type `polygon` with the coordinate system provided by the user and restricted to features within the bounding box provided. In the case that no value for *crs* is given the service returns the features in the coordinate system *defaultCRS*. Moreover, in the case that no value for *bb* is provided the service return all the features available. The execution of this function does not alter the state of the service.

Let *Building* and *Building*′ be resp. the state before and after the execution of the operation, the function *overlaps*:`bbox` $\times$ `bbox` $\rightarrow$ `boolean`, and the function *within*:`bbox` $\times$ `bbox` $\rightarrow$ `boolean`. The description of the function can be formalized in the following way:

$$\text{getBuildings} : \langle\ \text{crs} : \texttt{string},\ \text{bb} : \texttt{bbox}\ \rangle \rightarrow \langle\ \text{bldg} : \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle\ \rangle$$

Table 5.1: BNF for preconditions and postconditions

<expression> ::= <term> <relation> <term>
| <predicate>(<term>,...,<term>)
| ¬<expression>
| (<expression>)
| <expression> <connective> <expression>
| <quantifier> <variable> "|" <expression>
<term> ::= <function>(<term>,...,<term>)
| <constant>
| <variable>
<connective> ::= ∧ | ∨ | ⇒ | ⇔
<relation> ::= > | < | ⩾ | ⩽ | = | ≠ | ∈
<quantifier> ::= ∀ | ∃
<variable> ::= <state>
| <IO variable>
| <IO metadata>
| <service metadata>
<IO variable> ::= <input>|<output>|<input>.<attribute>|<output>.<attribute>
<IO metadata> ::= <IO variable>↑<metadata attribute>
<service metadata> ::= <state>↑<metadata attribute>

$$
\begin{aligned}
P \equiv\ & crs \neq null \Rightarrow crs \in Building{\uparrow}availableCRS \wedge \\
& bb \neq null \Rightarrow \text{overlaps}(bbox, Building{\uparrow}layerBBOX) \\
Q \equiv\ & Building' = Building \wedge \\
& crs \neq null \Rightarrow bldg.geom{\uparrow}crs = crs \wedge \\
& crs = null \Rightarrow bldg.geom{\uparrow}crs = Building{\uparrow}defaultCRS \wedge \\
& bb \neq null \Rightarrow \text{within}(bldg.geom{\uparrow}bbox, bb) \wedge \\
& bb = null \Rightarrow bldg.geom{\uparrow}bbox = Building{\uparrow}layerBBOX
\end{aligned}
$$

⌟

A common precondition on the state of a service tests whether an object with a particular identifier exists in the data source. For example, the operations GetFeatureById, DeleteFeature, UpdateFeature, InsertFeature in a WFS depend on the existence test of an identifier. In this way, let *ServiceFeatures* be the internal state variable about the features, we specify the precondition for an InsertFeature operation as:

$$\neg\exists\ f \in ServiceFeatures \mid ftr.id = f.id.$$

For variables of numeric types (`real` and `integer`) there exist two common patterns in pre- and postconditions. The first is to test whether such a variable's value is between two numbers. For

example, the function *ST_Line_Interpolate_Point*, defined in PostGIS [81], receives two inputs, a feature collection *ftr* with geometry `linestring`, and a real number *a* between `0` and 1. We can specify this requirement as the precondition $0 \leqslant a \wedge a \leqslant 1$.

The second pattern is used in testing a unit of measurement. Any numeric variable can be associated with a metadata attribute stating the unit of measurement. For example, let *a* be a `real` value the its unit of measurement can be accessed by $a{\uparrow}uom$. The unit of measurement is of type `string` and must follow the specification Quantities, Units, Dimensions and Data Types Ontologies (QUDT) [82]. An example value is "http://qudt.org/1.1/vocab/unit/Meter".

For a set typed variable value, we can use universal and existential quantifiers to specify conditions about its members. For example, a restriction that a set S of inner type `real` needs to be less than 100 is denoted as $\forall x \in S \mid x < 100$.

Next, we define metadata attributes that can be associated with a variable of geometry type:

- *crs*:`string` – The Coordinate Reference System (CRS) associated with the geometry. It complies with the URL convention of OGC Name Type Specification for Coordinate Reference Systems [83]. Examples of values are: "noCRS" (in the case no CRS is specified for the geometry), and "http://www.opengis.net/def/crs/OGC/1.3/CRS84".

- *dimension*:`array(string)` – The number of dimensions of the geometry must be provided. The OGC Simple Feature Access standard specifies the following dimensions: "x", "y", "z", "m", in such a way that a 2-dimensional geometry contains ["x", "y"] information, a 3-dimensional geometry contains ["x", "y", "z"] or ["x", "y", "m"], and 4-dimensional geometry contains ["x", "y", "z", "m"]. In this thesis, the number of measurement dimensions is not restricted to one, as discussed in Chapter 6. This metadata attribute specifies each of the used dimensions, including each measure. For example, a valid value is ["x", "y", "atemp", "depth"], where *atemp* and *depth* are measurement specified in the GPX TrackPointExtension V1 [84].

- *bbox*:`bbox` – The information about the bounding box of a feature or a set of features. It has the same dimension as the geometry. An example value is [3.3579, 50.7504, 7.2275, 53.5552]. In this case is specified the bottom left corner and the top right corner of the bounding box.

- *bboxFormat*:`string` – One can specify the bounding box in different formats. The above example uses the GeoJSON encoding. Examples values are "GeoJSON", "GML" (envelope object), and "WKT" (encoded as polygon).

- *bboxCrs*:`string` – The information of the coordinate system of the bounding box. The possible values follow the same convention as those for *crs*.

- *verticalDatum*:`string` – In case that a geometry has a dimension with a *z* dimension, a vertical datum needs to be indicated. The allowed values are described in the specification WKT representation of coordinate reference systems [85].

  An example value is "http://www.opengis.net/def/crs/vertical/normaalAmsterdamsPeil".

- *format*:`string` – Structural Composability verifies the formats of the transferred files, however even for a single file format, one can use different geometry formats. Possible values are: "WKT", "WKB", "EWTK", "EWKB", "TWKB", and "GeoJSON".

Variables of type `bbox` can be associated with three metadata attributes : *crs*, *format*, and *dimension*. They behave in the same way as *bboxCrs*, *bboxFormat*, *dimension*, respectively.

Coverage metadata used to verify this level of composability is based on the GML Application Schema — Coverages [77]. We list the following :

- *quantityDefinition*:`string` – Definition of what the value of the pixel represents. It must follow the URL format defined in the GML Application Schema — Coverages [77]. An example value is: "http://opengis.net/def/property/OGC/0/Radiance".

- *uom*:`string` Unit in which the pixel value is measured. It is defined in the same manner as in numeric types.

- *valueSpace*:`array(real)` – Defines the range of acceptable pixel values. It is an array with exactly two elements. A example of a value is [0,255]. The value used for no data pixels is not included in this range.

- *noData*:`real` – Value used to encode no data (null) values. Example: -9999.0.

- *cellSizeX*:`real` – and *cellSizeY*:`real` In case of RectifiedGridCoverage and GridCoverage, the cell size in the X and Y direction are specified. They apply the same unit as the CRS. Example: 10.0.

A variable of type coverage can also be associated with the metadata for *crs*, *bbox*, *bboxFormat*, and *bboxCrs*, all defined in the same way as in geometry types.

Chapter 4 defines the temporal types used in the type system. The temporal metadata used to verify this level of composability is based on the concepts of the current work of OGC Technical Committee's Temporal Domain Working Group [86]. We define the following metadata attributes for temporal data:

- *format*:`string` – Format in which time is represented, with allowed values described in ISO 8601 [74] and in the candidate standard for a Temporal WKT [86]. Examples values are: "YYYY-MM", "YYYY", "YY", "YYYY-DDD", "YYYY-Www", "YYYY-Www-D", and "YYYY-MM-DDThh:mm:ss", where *W* indicates the week designator, and *T* indicates the time designator.

- *trs*:`string` – The Temporal Reference System (TRS) in which time is measured. It can be a calendar or an ordinal scale. Based on [86], possible values for a calendar are "julian", "gregorian", "prolepticGregorian", "hijriCalendar", and for a ordinal scale, "geologic".

- *epoch*:`string` – The time instant of the beginning of a calendar. For example, Unix Time [87] is defined as the number of seconds since 1 January of 1970, ignoring leap seconds. Examples values are: "commonEra", "unixTime", and "julianDay".

- *timeZone*:`string` – When one uses Coordinated Universal time (UTC), there is a need of specifying offsets to convert to local time. The values are based on ISO 8601. The possible values are: *"Z"*, when the time is expressed in UTC; and values following the pattern *+hh:mm* or *-hh:mm* when times are expressed in local time, for example, for Central European Time (CET) the value is "+01:00". Daylight saving time is also informed based on timeZone.

Based on the metadata attributes presented above we introduce the notion of metadata type.

Definition 5.3 (Metadata type)**.** Input and output identifiers of service types can be associated with a secondary type, the *metadata type*. This type indicates the metadata attributes available to the particular input or output identifier. Let t be an input or output identifier, and $m_1 : \tau_1, \ldots, m_k : \tau_k$ be metadata attributes as defined in this section with their correspondent type. The metadata type for t is denoted by:

$$t\uparrow : \langle m_1 : \tau_1, \ldots, \ m_n : \tau_k \rangle \tag{5.1}$$

In this way the notation for an attribute selection would be $t\uparrow.m_k$, which we simplify as $t\uparrow m_k$.

Services that passed a type checking but fail a metadata type-checking are still considered Static Syntactic Composable. However, for those services is not possible to evaluate the Dynamic Syntactic Composability.

5.2.2 Relaxed Conditions

We may conclude from the above that one cannot possibly verify all the above conditions at compile-time, especially conditions that relate to the state of the service. In the context of compile-time verification, we propose that the precondition is relaxed to contain variables only that can be evaluated. This means that we cannot ensure the correct execution of the service, and must require additional run-time verification. Also, at compile-time, since actual data is not available, we use the postcondition of the process as a description of the data characteristics. In this way, to verify a composition at compile-time, we need a mechanism that evaluates the precondition of a service based on the postconditions of its inputs.

Hoare [19] defines the rules of composition and consequence. The first gives a criterion for reasoning about the correctness of sequential execution of two instructions. The second specifies how one can modify the precondition and postcondition to obtain either a weaker postcondition or stronger precondition.

$$\frac{\{P\}\ S\ \{R\}, \{R\}\ T\ \{Q\}}{\{P\}\ S;T\ \{Q\}} \qquad \text{(Rule of Composition)}$$

$$\frac{P_1 \Rightarrow P_2, \{P_2\}\ S\ \{Q_2\}, Q_2 \Rightarrow Q_1}{\{P_1\}\ S\ \{Q_1\}} \qquad \text{(Rule of Consequence)}$$

The rule of consequence is of particular importance for the verification of Service Substitutability. A service S' (with triple $\{P'\}S'\{Q'\}$) can substitute a service S (with triple $\{P\}S\{Q\}$) it is necessary that $P \Rightarrow P'$ and $Q' \Rightarrow Q$. Section 5.3 further discusses service substitutability.

By the combination of both the rule of composition and the rule of consequence, we can derive a stronger composition rule:

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, \{P_2\}\ S_2\ \{Q_2\}, Q_1 \Rightarrow P_2}{\{P_1\}\ S_1; S_2\ \{Q_2\}} \tag{5.2}$$

In this way to proof that a composition between two service is valid, we need to proof the assertion $Q_1 \Rightarrow P_2$ given that Q_1 holds. We exemplify five rules of inference that can be used to prove this assertion.

$$\frac{\text{TRIVIAL CASE}\quad A \qquad B}{A \wedge B \Rightarrow B} \tag{5.3}$$

ASSERTION ELIMINATION

$$\frac{A \qquad B}{(A \wedge B \Rightarrow C \wedge B) \Leftrightarrow (A \Rightarrow C)} \tag{5.4}$$

EXISTENTIAL ELIMINATION

$$\frac{\exists x | A(x) \qquad B}{(\ (\ \exists x | A(x) \wedge B\ ) \Rightarrow C\ ) \Leftrightarrow (B \Rightarrow C)} \tag{5.5}$$

VALUE SUBSTITUTION

$$\frac{x = v \qquad A(x)}{(\ (x = v) \wedge A(x) \Rightarrow B(x)) \Leftrightarrow (A(v) \Rightarrow B(v))} \tag{5.6}$$

OR PROOF

$$\frac{A_1 \vee \cdots \vee A_n}{(A_1 \vee \cdots \vee A_n \Rightarrow B) \Leftrightarrow (\ (A_1 \Rightarrow B) \wedge \cdots \wedge (A_n \Rightarrow B)\ )} \tag{5.7}$$

Aside from the mechanism for verifying the precondition of a service based on the postcondition of its inputs, we also need to define the notion of a relaxed condition.

Definition 5.4 (Relaxed condition)**.** Let R be an precondition or a postcondition of a service, the *relaxed condition* is defined as the result of applying to R existential quantification to all variables that the value is not known at compile-time. In this case the variables refer to input, output, or state, since IO metadata and service metadata are required to be known at compile-time. Let $x_1 : \tau_1, \ldots, x_n : \tau_n$ be variables with value not known at compile-time and their correspondent types. The relaxed condition is denoted as:

$$\widetilde{R} \equiv \exists x_1 : \tau_1 \ \ldots \ \exists x_n : \tau_1 \mid R \tag{5.8}$$

Ideally, after we apply the existential quantifier, the relaxed condition can be further simplified.

Example 5.2. Consider a service that uses the operation *insertRivers*. It receives one input, *ftr* : $\langle$id : `string`, geom : `linestring`$\rangle$, which needs to use crs EPSG:28992, must overlap a given bounding box, and there should not exist the same identifier in the current state of the service. Also the metadata type for *ftr* is *ftr*$\uparrow$: $\langle$crs : `string`$\rangle$. Let *Rivers* and *Rivers*$'$ respectively be the state before and after the execution of the operation. We define the pre- and postconditions as follows:

$$\begin{aligned} P \equiv\ & ftr.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & \text{overlaps}(ftr.geom,\ [3.3579, 50.7504, 7.2275, 53.5552]) \wedge \\ & \neg\exists\, rvr \in Rivers \mid ftr.id = rvr.id \\ Q \equiv\ & Rivers' = Rivers \cup ftr \end{aligned}$$

Since at compile-time, it is not possible to verify identifier non-existence, and the geometry is not known the relaxed precondition becomes:

$$\begin{aligned} \widetilde{P} \equiv \exists ftr : \mid\ & ftr.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & \text{overlaps}(ftr.geom,\ [3.3579, 50.7504, 7.2275, 53.5552]) \wedge \\ & \neg\exists\, rvr \in Rivers \mid ftr.id = rvr.id \end{aligned}$$

Let $R \equiv ftr.geom{\uparrow}crs =$ “http://www.opengis.net/def/crs/EPSG/0/28992”, we have that $R \Rightarrow \widetilde{P}$, therefore is a weaker (relaxed) precondition. In this way at compile-time we just evaluate the precondition R. Moreover, naturally, we need to perform also a run-time verification of Dynamic Semantic Composability.

The condition $Q_1 \Rightarrow P_2$ suffices to ensure composition validity; however it is not a necessary condition. Even when $Q_1 \not\Rightarrow P_2$, the two services can compose a valid service. In Chapter 4, we also discussed the existence of false negatives in Static Syntactic Composability. Also, the existence of false positives inherently comes with compile-time verification. As stated before, we cannot verify all preconditions at compile-time, such as those related to the state of the service. Therefore, at compile-time, we consider a relaxed precondition that only takes into account terms that are verifiable at compile-time. With this relaxed precondition, we cannot ensure the absence of run-time errors.

The composition in Equation (5.2) has one shortcoming in the analyses of web service composition correctness: we cannot reason about value hand-over between the output of one service and the input of the next service in their composition. To derive a stronger result, Section 5.2.3 defines a relation between the Hoare triple and the inputs and outputs of the service.

5.2.3 Inputs and Outputs

In this section, we extend the instruction notation to represent explicitly the inputs and outputs of a service and develop methods to manipulate pre- and postconditions based on the interaction between services. We use the same notation for inputs and outputs of a service introduced in Chapter 4, with the modification that we can omit types. Moreover, we extend the notation to allow optional inputs, which may be associated with a default value. Optional inputs are represented with the notational convention ${}^{\star}i \text{ <-- } d$, where the left superscript $\star$ indicates the input is optional, and d is the default value associated. Below, in the case that the inputs or outputs are not interesting for a particular example they are omitted. The inputs of the service C with optional input i_k and associated default value d is denoted as $C : \langle i_1, \ldots, {}^{\star}i_k \text{ <-- } d, \ldots \rangle \rightarrow \langle \ldots \rangle$.

We denote a service composition by indicating the component services and the value hand over indication. Here, *value hand over indication* is a relationship between an output identifier of a service and the input identifier of another. At run-time this indication realizes into data transferring between the services. Let C_1 and C_2 be services such that $C_1 : \langle \ldots \rangle \rightarrow \langle o_1, \ldots \rangle$ and $C_2 : \langle i_1, \ldots \rangle \rightarrow \langle \ldots \rangle$, the notation $C_2; C_1(o_1 \leftarrow i_1)$ indicate that the services C_1 and C_2 are composed with a value hand over indication between o_1 and i_1.

In this section, we present a series of term rewriting rules that allow the verification of the composability when the composition includes multiple services, and different value hand over patterns between them. In the case of a composition of two services, by Equation (5.2) we need to evaluate the implication relation between the postcondition of the first service and the precondition of the second service. Naturally both services will use different variable names to define their pre- and postconditions, and first we need to rename the variable names accordingly to the value hand over interaction. The rule *Assertion alias* performs the variable renaming.

Definition 5.5 (Assertion alias)**.** Let $C_2; C_1(o_1 \leftarrow i_1, \ldots o_n \leftarrow i_n,)$ be a composition between two services, with hand over of data from C_1 to C_2, and let $\{P\}\, C_1 \,\{Q\}$. The assertion alias rename the variables of Q based on the value hand over indication with C_2 is denoted by $[o_1/i_1; \ldots; o_n/i_n]Q$.

This renamed postcondition is used for the evaluation of composability.

Example 5.3. Let *getAccidents* and *computeConvexHull* be services such that $getAccidents : \langle \ldots \rangle \rightarrow \langle \text{pnt} \rangle$, and $computeConvexHull : \langle \text{ftr} \rangle \rightarrow \langle \ldots \rangle$, where *pnt* and *ftr* are feature collections with a geometry *geom*. Figure 5.1 shows the relation between inputs and outputs of services *getAccidents* and *computeConvexHull*.

The postcondition for *getAccidents* is:

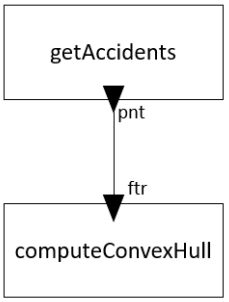


Figure 5.1: Simple relation between inputs and outputs of two services

$$
\begin{aligned}
Q_1 \equiv \quad & pnt.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\
& pnt.geom{\uparrow}bbox = [5.12, 52.77, 5.53, 53.24] \wedge \\
& pnt.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\
& pnt.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\
& pnt.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}]
\end{aligned}
$$

The precondition for *computeConvexHull* is:

$$P_2 \equiv ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}]$$

The composition is represented by getAccidents; computeConvexHull($ftr \leftarrow pnt$). To this composition be valid, is sufficient (by equation 5.2) that $Q_1 \Rightarrow P_2$. Both conditions use different variables, being necessary to rename them before evaluate the composability. Based on the interaction between the input and output we apply the assertion alias on Q_1:

$$
\begin{aligned}
& ftr.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\
& ftr.geom{\uparrow}bbox = [5.12, 52.77, 5.53, 53.24] \wedge \\
& ftr.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\
& ftr.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\
& ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}]
\end{aligned}
$$

Now the variable names are consistent, and we can evaluate the precondition. By Equation 5.3 we have that the services are composable.

⌟

Next, we present the conjunction rule. It defines how to assess a precondition when connections exist that arise from outputs of multiple services.

Definition 5.6 (Conjunction)**.** Let $C_0, \ldots, C_w$ be services in a composition such that $C_1, \ldots, C_w$ hand over values to C_0. And consider the Hoare triples $\{P_k\}\, C_k\, \{Q_k\}$ for $k = 1, \ldots, w$. To verify this composition is necessary to derive a combined postcondition for the services $C_1, \ldots, C_w$. The *conjunction rule* defines that combined postcondition used for the evaluation of the composability is:

$$Q_1 \wedge Q_2 \wedge \cdots \wedge Q_w \tag{5.9}$$

Example 5.4. Let *getRoads*, *getNeighbourhoods*, and *clipFeatures* be services such that:

$$getRoads : \langle \ldots \rangle \rightarrow \langle \text{rd} : \mathbb{P}\langle geom : \texttt{linestring} \rangle \rangle$$
$$getNeighbourhoods : \langle \ldots \rangle \rightarrow \langle \text{ftr} : \mathbb{P}\langle geom : \texttt{polygon} \rangle \rangle$$
$$clipFeatures : \langle \text{ftr} : \mathbb{P}\langle geom : \texttt{geometry} \rangle, \text{clipper} : \mathbb{P}\langle geom : \texttt{polygon} \rangle \rangle \rightarrow \langle \ldots \rangle$$

Figure 5.2 shows the relation between inputs and outputs of services *getRoads*, *getNeighbourhoods* and *clipFeatures*.

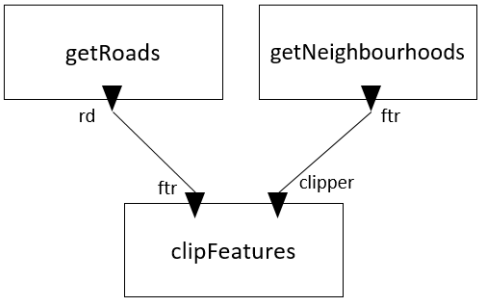


Figure 5.2: Composition case with a conjunction of two services

The postcondition for *getRoads* is:

$$\begin{aligned} Q_1 \equiv \quad & rd.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & rd.geom{\uparrow}bbox = \text{“POLYGON((3.2 50.7, 7.01 50.7, 7.01 53.5, 3.2 53.5, 3.2 50.7))”} \wedge \\ & rd.geom{\uparrow}bboxFormat = \text{“WKT”} \wedge \\ & rd.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & rd.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}] \end{aligned}$$

The postcondition for *getNeighbourhoods* is:

$$\begin{aligned} Q_2 \equiv \quad & ftr.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & ftr.geom{\uparrow}bbox = [3.15, 51.12, 5.96, 53.03] \wedge \\ & ftr.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\ & ftr.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}] \end{aligned}$$

The precondition for *clipFeatures* is:

$$\begin{aligned} P_3 \equiv \quad & ftr.geom{\uparrow}crs = clipper.geom{\uparrow}crs \wedge \\ & ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}] \wedge \\ & clipper.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}] \end{aligned}$$

The composition is represented by:

$$\text{getRoads}; \text{getNeighbourhoods}; \text{clipFeatures}\langle ftr \leftarrow rd,\ clipper \leftarrow ftr \rangle$$

To verify the precondition of *clipFeatures* we need to combine the postcondition of *getRoads* and *getNeighbourhoods*, which is done by the conjunction rule.

$ftr.geom{\uparrow}crs =$ "http://www.opengis.net/def/crs/EPSG/0/4326" $\wedge$
$ftr.geom{\uparrow}bbox =$ "POLYGON((3.2 50.7, 7.01 50.7, 7.01 53.5, 3.2 53.5, 3.2 50.7))" $\wedge$
$ftr.geom{\uparrow}bboxFormat =$ "WKT" $\wedge$
$ftr.geom{\uparrow}bboxCrs =$ "http://www.opengis.net/def/crs/EPSG/0/4326" $\wedge$
$ftr.geom{\uparrow}dimension =$ ["x", "y"] $\wedge$
$clipper.geom{\uparrow}crs =$ "http://www.opengis.net/def/crs/EPSG/0/28992" $\wedge$
$clipper.geom{\uparrow}bbox = [3.15, 51.12, 5.96, 53.03] \wedge$
$clipper.geom{\uparrow}bboxFormat =$ "GeoJSON" $\wedge$
$clipper.geom{\uparrow}bboxCrs =$ "http://www.opengis.net/def/crs/EPSG/0/4326" $\wedge$
$clipper.geom{\uparrow}dimension =$ ["x", "y"]

With the combined postcondition we can evaluate if the composition is valid. By Equations 5.4 and 5.6 we verify that the composition is not valid, due to differences in the coordinate system.

⌟

One special case of assertion alias is when the composition has more than one connection from a single output. In this case, the direct application of the alias rule would create a conflict of variable names. To solve this case we introduce the *Postcondition copy* rule.

Definition 5.7 (Postcondition copy)**.** Let C_1 and C_2 be services with $\{P_1\}\, C_1\, \{Q_1\}$, $C_1 : \langle \ldots \rangle \rightarrow \langle o_1, \ldots \rangle$, and $C_2 : \langle i_1, \ldots \rangle \rightarrow \langle \ldots \rangle$. In the case of a composition represented by $C_1; C_2 \langle i_1 \leftarrow o_1, i_2 \leftarrow o_1, \ldots \rangle$, since the output o_1 is used twice, the renaming of the variables is performed in the following way:

$$[o_1/i_1; o_1/i_2; \ldots]Q_1 \equiv [o_1/i_1; \ldots]Q_1 \wedge [o_1/i_2; \ldots]Q_1 \tag{5.10}$$

Example 5.5. Let *getSatelliteImg*, *getDisplacements*, and *correctGeo* be services such that:

$$\begin{aligned} \text{correctGeo} :& \langle \text{raster} : \texttt{rectifiedgridcoverage},\ \text{ref} : \texttt{rectifiedgridcoverage}, \\ & \quad \text{vectors} : \mathbb{P}\langle \text{geom} : \texttt{linestring} \rangle \rangle \rightarrow \langle \ldots \rangle \\ \text{getSatelliteImg} :& \langle \ldots \rangle \rightarrow \langle \text{img} : \texttt{rectifiedgridcoverage} \rangle \\ \text{getDisplacements} :& \langle \ldots \rangle \rightarrow \langle \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{linestring} \rangle \rangle \end{aligned}$$

Figure 5.3 shows the relation between inputs and outputs of services *getSatelliteImg*, *getDisplacements*, and *correctGeo*.

The service *correctGeo* receives as an input a raster and refines its georeference (better positional quality), a reference grid used to resample the georeferenced raster, and a set of vectors which represent the control vectors to improve the raster planimetric accuracy. This example service is based on the transformer RubberSheeter from FME [88]. The postcondition for *getSatelliteImg* is:

$$\begin{aligned} Q_1 \equiv \quad & img{\uparrow}crs = \text{"http://www.opengis.net/def/crs/EPSG/0/28992"} \wedge \\ & img{\uparrow}cellSizeX = 5 \wedge \\ & img{\uparrow}cellSizeY = 5 \end{aligned}$$

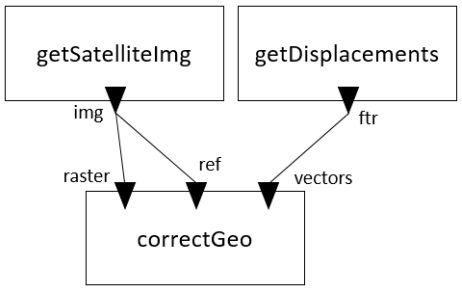


Figure 5.3: Composition case where a postcondition copy is needed

The *getDisplacements* feature service returns a set of 2-points `linestring` lines, where the start point represents the original coordinates and the end point the corrected coordinates. The postcondition for *getDisplacements* is:

$$Q_2 \equiv \quad ftr.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ ftr.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”}]$$

The precondition for *correctGeo* is:

$$P_3 \equiv \quad raster{\uparrow}crs = vectors.geom{\uparrow}crs \wedge \\ raster{\uparrow}crs = ref{\uparrow}crs \wedge \\ vectors.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}]$$

The composition is denoted by:

$$\text{getSatelliteImg; getDisplacements; correctGeo}\langle raster \leftarrow img,\ ref \leftarrow img,\ vectors \leftarrow ftr\rangle$$

Since is a repetitive use of the output *img*, we need to apply the postcondition copy rule:

$$raster{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ raster{\uparrow}cellSizeX = 5 \wedge \\ raster{\uparrow}cellSizeY = 5 \wedge \\ ref{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ ref{\uparrow}cellSizeX = 5 \wedge \\ ref{\uparrow}cellSizeY = 5$$

To verify the composition is still needed to apply the conjugation operation.

$$raster{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ raster{\uparrow}cellSizeX = 5 \wedge \\ raster{\uparrow}cellSizeY = 5 \wedge \\ ref{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ ref{\uparrow}cellSizeX = 5 \wedge \\ ref{\uparrow}cellSizeY = 5 \wedge \\ vectors.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ vectors.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”}]$$

By Equations 5.4 and 5.6 we verify that the composition is valid.

⌟

The Assertion alias rule is the basis for a consistent evaluation of a composition, which allows a meaningful link between the free variables of the postcondition and precondition. However not necessarily all outputs of one service will be used as input to the next service in the composition. To remove variables that are not part of the value hand over of the postcondition we define the *Postcondition projection* rule.

Definition 5.8 (Postcondition projection)**.** Let C be a service such that $\{P\}C\{Q\}$, and the outputs are:

$$C : \langle \ldots \rangle \rightarrow \langle o_1 : \tau_1, \ldots, o_n : \tau_n \rangle$$

And let $K = \{o_{k_1}, \ldots, o_{k_m}\}$ be a subset of the outputs of C. The projection of the postcondition Q with scope K is denoted by:

$$\exists o_{k_1} : \tau_{k_1} \ldots \exists o_{k_m} : \tau_{k_1} \mid Q \tag{5.11}$$

Example 5.6. Let *intersectFeatures* and *calculateArea* be services such that:

$$\begin{aligned} \text{intersectFeatures} : &\langle \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{geometry} \rangle,\ \text{filter} : \mathbb{P}\langle \text{geom} : \texttt{geometry} \rangle \rangle \\ &\rightarrow \langle \text{passed} : \mathbb{P}\langle \text{geom} : \texttt{geometry} \rangle,\ \text{failed} : \mathbb{P}\langle \text{geom} : \texttt{geometry} \rangle \rangle \\ \text{calculateArea} : &\langle \text{ply} : \mathbb{P}\langle \text{geom} : \texttt{polygon} \rangle \rangle \rightarrow \langle \ldots \rangle \end{aligned}$$

Figure 5.4 shows the relation between inputs and outputs of services *intersectFeatures* and *calculateArea*.

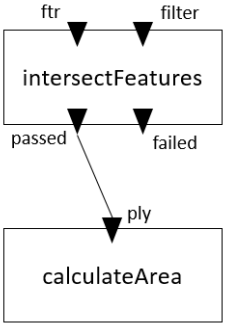


Figure 5.4: Composition case where postcondition projection is needed

The postcondition for *intersectFeatures* is:

$$\begin{aligned} Q \equiv\ & passed.geom{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & failed.geom{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & passed.geom{\uparrow}dimension = ftr.geom{\uparrow}dimension \wedge \\ & failed.geom{\uparrow}dimension = ftr.geom{\uparrow}dimension \wedge \\ & \text{within}(passed.geom{\uparrow}bbox,\ filter.geom{\uparrow}bbox) \wedge \\ & \text{within}(failed.geom{\uparrow}bbox,\ ftr.geom{\uparrow}bbox) \end{aligned}$$

The composition is represented by intersectFeatures; calculateArea$\langle ply \leftarrow passed \rangle$. Applying postcondition projection and assertion alias the postcondition becomes:

$$\begin{aligned} \exists\, failed : \mathbb{P}\langle \text{geom} : \texttt{geometry} \rangle \mid \quad & ply.geom{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & failed.geom{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & ply.geom{\uparrow}dimension = ftr.geom{\uparrow}dimension \wedge \\ & failed.geom{\uparrow}dimension = ftr.geom{\uparrow}dimension \wedge \\ & \text{within}(ply.geom{\uparrow}bbox, filter.geom{\uparrow}bbox) \wedge \\ & \text{within}(failed.geom{\uparrow}bbox, ftr.geom{\uparrow}bbox) \end{aligned}$$

Which, by the existential elimination (equation 5.5), is simplified to:

$$\begin{aligned} & ply.geom{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & ply.geom{\uparrow}dimension = ftr.geom{\uparrow}dimension \wedge \\ & \text{within}(ply.geom{\uparrow}bbox, filter.geom{\uparrow}bbox) \end{aligned}$$

⌟

Remark. The goal of postcondition projection is to bind the variables that are not part of the composition.

One not necessarily needs to fill all the inputs of a service. The input can be optional, with a default value, or completely omitting its use. In the latter case, one needs to variables of the precondition that are not part of the value hand over. To accomplish this goal, we define the *Precondition projection* rule.

Definition 5.9 (Precondition projection)**.** Let C be a service with $\{P\}\ C\ \{Q\}$, and:

$$P : \langle i_1 : \tau_1, \ldots, i_n : \tau_n \rangle \rightarrow \langle \ldots \rangle$$

And let $K = \{i_{k_1}, \ldots, i_{k_m}\}$ be a subset of the inputs of C. The projection of the precondition P with the scope K is denoted by:

$$\exists i_{k_1} : \tau_{k_1} \ldots \exists i_{k_m} : \tau_{k_m} \mid P \tag{5.12}$$

Example 5.7. Let *getEmergencyCenters* and *computeVoronoi* be services such that:

$$\begin{aligned} \text{computeVoronoi} &: \langle \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{point} \rangle,\ {}^{\star}\text{bb} : \texttt{bbox} \rangle \rightarrow \langle \ldots \rangle \\ \text{getEmergencyCenters} &: \langle \ldots \rangle \rightarrow \langle \text{centers} : \mathbb{P}\langle \text{geom} : \texttt{point} \rangle \rangle \end{aligned}$$

Figure 5.5 shows the relation between inputs and outputs of services *getEmergencyCenters* and *computeVoronoi*.

The postcondition for *getEmergencyCenters* is:

$$\begin{aligned} Q_1 \equiv \quad & centers.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & centers.geom{\uparrow}bbox = [3.3421, 50.7666, 5.132, 51.0012] \wedge \\ & centers.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\ & centers.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & centers.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”}] \end{aligned}$$

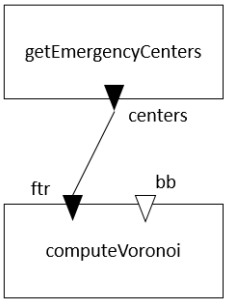


Figure 5.5: Composition case with an optional input

The *computeVoronoi* service can receive a bounding box for the Voronoi diagram calculation. If it does not receive such, it will calculate an appropriate bounding box. The precondition for *computeVoronoi* is:

$$\begin{aligned} P_2 \equiv \quad & \text{within}(ftr.geom{\uparrow}bbox, bb) \wedge \\ & bb{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}] \wedge \\ & bb{\uparrow}format = \text{“GeoJSON”} \end{aligned}$$

The composition is represented by:

$$\text{getEmergencyCenters}; \text{computeVoronoi}\langle ftr \leftarrow centers, \ ^{\star}bb\rangle$$

Since no value was given to $^{\star}bb$, for the evaluation of the composition the precondition of *computeVoronoi* has to be projected, becoming:

$$\begin{aligned} \exists\, bb : \texttt{bbox} \mid \quad & \text{within}(ftr.geom{\uparrow}bbox, bb) \wedge \\ & bb{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}] \wedge \\ & bb{\uparrow}format = \text{“GeoJSON”} \end{aligned}$$

Which by the existential elimination (equation 5.5) is simplified to:

$$ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}]$$

Moreover, by equation 5.3 we have that the services are composable.

⌟

The case of optional input where the default information is available is simpler and does not require precondition projection.

Example 5.8. Consider the service $buffer : \langle \text{ftr}, \ \text{distance}, \ ^{\star}\text{unit} \leftarrow\!\!- \text{“meters”}\rangle \rightarrow \langle \ldots \rangle$ and the *getEmergencyCenters* service described in the last example. In the case of a composition:

$$\text{getEmergencyCenters}; \text{buffer}\langle ftr \leftarrow centers, \ distance \leftarrow 500, ^{\star}unit\rangle$$

Since no value was given to *unit* and a default value exists, the composition is represented by:

$$\text{buffer}\langle ftr \leftarrow centers, \ distance \leftarrow 500, \ unit \leftarrow \text{“meters”}\rangle$$

⌟

So far, in all defined compositions, the inputs are bound to a single output, which is not true in all cases. The rule *Precondition copy* addresses the case of multiple values hand over to a single input. This rule is applied before the alias rule to avoid conflict of variable names.

Definition 5.10 (Precondition copy)**.** Let C be a service with $\{P\}C\{Q\}$ and $C : \langle i_1, \dots\rangle \rightarrow \langle \dots\rangle$, and let v_1 and v_2 be valid inputs. In the case of a composition $C\langle i_1 \leftarrow v_1, i_1 \leftarrow v_2, \dots\rangle$, to evaluate the composition we consider the precondition as:

$$[i_1/_1 i_1]P \wedge [i_1/_2 i_1]P \tag{5.13}$$

Where ${}_1i_1$ and ${}_2i_1$ are auxiliary variables. In addition, the composition relation is rewritten as $C\langle {}_1i_1 \leftarrow v_1, {}_2i_1 \leftarrow v_2, \dots\rangle$

Example 5.9. Consider the services *getVegetation*, *getForest* and *dissolveFeatures* such that:

$$\begin{aligned} \text{dissolveFeatures} &: \langle \text{ply} : \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle\rangle \rightarrow \langle \dots\rangle \\ \text{getVegetation} &: \langle \dots\rangle \rightarrow \langle \text{veg} : \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle\rangle \\ \text{getForest} &: \langle \dots\rangle \rightarrow \langle \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{polygon}\rangle\rangle \end{aligned}$$

Figure 5.6 shows the relation between inputs and outputs of services *getVegetation*, *getForest* and *dissolveFeatures*.

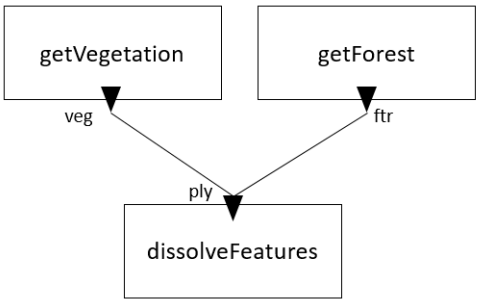


Figure 5.6: Composition case where a precondition copy is needed

The precondition for *getVegetation* is:

$$\begin{aligned} Q_1 \equiv \quad & veg.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & veg.geom{\uparrow}bbox = [3.3, 50.7, -1.3, 5.1, 51.3, 17.4] \wedge \\ & veg.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\ & veg.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & veg.geom{\uparrow}verticalDatum = \text{“.../normaalAmsterdamsPeil”} \wedge \\ & veg.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}, \text{“z”}] \end{aligned}$$

The precondition for *getForest* is:

$$\begin{aligned} Q_2 \equiv \quad & ftr.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & ftr.geom{\uparrow}bbox = [3.1, 50.6, 3.85, 51.9] \wedge \\ & ftr.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\ & ftr.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & ftr.geom{\uparrow}dimension = [\text{“x”}, \text{“y”}] \end{aligned}$$

And the precondition for *dissolveFeatures* is:

$$P_3 \equiv \quad ply.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”}]$$

The composition is represented by

$$\text{getVegetation; getForest; dissolveFeatures}\langle ply \leftarrow veg,\ ply \leftarrow ftr\rangle$$

Since there is a repetitive use of the input ply there is a need of precondition copy, which alters the relation between inputs and outputs to:

$$\text{dissolveFeatures}\langle ply_1 \leftarrow veg, ply_2 \leftarrow ftr\rangle$$

And the precondition to the evaluation of the composition becomes:

$$ply_1.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”}] \wedge$$
$$ply_2.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”}]$$

And by applying the conjunction rule the combined postcondition of *getVegetation* and *getForest* becomes:

$$ply_1.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge$$
$$ply_1.geom{\uparrow}bbox = [3.3, 50.7, -1.3, 5.1, 51.3, 17.4] \wedge$$
$$ply_1.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge$$
$$ply_1.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge$$
$$ply_1.geom{\uparrow}verticalDatum = \text{“.../normaalAmsterdamsPeil”} \wedge$$
$$ply_1.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”},\ \text{“z”}] \wedge$$
$$ply_2.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge$$
$$ply_2.geom{\uparrow}bbox = [3.1, 50.6, 3.85, 51.9] \wedge$$
$$ply_2.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge$$
$$ply_2.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge$$
$$ply_2.geom{\uparrow}dimension = [\text{“x”},\ \text{“y”}]$$

Which by Equation 5.6 is not a valid composition, due to the invalid dimension in the input *getVegetation*.

⌟

In practice, the presented rules are used in combination. If f and g are rules, one needs to verify whether $f \circ g$ is valid and whether $f \circ g = g \circ f$. This helps to understand when and how to apply those rules in the case where one requires multiple rules.

Only in three cases of $f \circ g$ is not valid: Alias then Postcondition Copy; Alias then Precondition Copy; and Postcondition Copy then Precondition Copy. They are not valid due to a conflict in variable names. As a general recommendation, the alias rule is performed last in all cases, and the precondition copy first. For all valid cases, it is trivially true that $f \circ g = g \circ f$.

5.2.4 Conditional, Subgraph, and Loop

Chapter 3 defines the conditional node with one input and two outputs, which creates an exclusive disjunction in the graph. It allows the evaluation of a boolean expression about the variables of its input at run-time. The boolean expression is assigned to the conditional node upon its creation,

and this conditional is evaluated once at run-time. In the case the expression evaluates to true, the output path *true* is chosen. In the case it evaluates to false, the path *false* is chosen. This operator does not behave in the same way as a service because its two output paths (*true* or *false*) have different postconditions.

Suppose $C : \langle i \rangle \rightarrow \langle true, false \rangle$ a conditional node with an associated boolean expression B, and an input S such that $\{P\}\ S\ \{Q\}$ with $S : \langle \ldots \rangle \rightarrow \langle o_1, \ldots \rangle$. And consider the composition $\mathrm{S;C}(i \leftarrow o_1)$, we define the postcondition of the output path *true* as $Q \wedge B$, and the postcondition of *false* as $Q \wedge \neg B$.

This is especially useful when a postcondition of a service is of the form $Q = Q_1 \vee Q_2$. In this case, one can associate the expression Q_1 with the conditional node, in which the postcondition of the composition in the output path *true* becomes:

$$Q \wedge Q_1 \equiv (Q_1 \vee Q_2) \wedge Q_1 \equiv Q_1$$

and for the output path *false* it becomes Q_2.

Chapter 3 also introduces subgraph nodes, a service node that can be associated with parametric subgraphs. Parametric graphs are graphs with one or more input parameter node, and one or more output parameter node. An input parameter node can be connected to one or more services in the subgraph. We define the postcondition of an input parameter node as the conjunction of the preconditions of all connected services. The reason is that the input parameter must be handover a valid value for all connected services, meaning that it must fulfill all the preconditions simultaneously.

An output parameter node can be connected to only one service in the subgraph. We define the precondition of the output parameter as the postcondition of the connected service since the service that is connected to it must hand over a valid value. We define the precondition of the subgraph node as the postcondition of the input parameter node. Moreover, the postcondition of the subgraph node as the precondition of the output parameter node.

Chapter 3 also defines loop nodes. The loop nodes are also associated with parametric graphs, and the pre- and postcondition definition behaves in a similar way as described for subgraph nodes. For the loop Iterate Inputs, we can define the precondition in the same way as in subgraph. However, the postcondition is defined per output (such as in a conditional), and each of the postconditions is the precondition of the correspondent output parameter node.

For the loop iterate sets, let I_s be a iterate multivalue node such as $I_s : \langle i \rangle \rightarrow \langle o \rangle$, where i and o are sets. The correspondent input parameter to i in the parametric graph only refers to one element of the set. Let T be the postcondition of the of the input parameter. The precondition for the entire set is $\forall x \in i \mid T$. Let R be the precondition of the output parameter correspondent to o, it also only refers to one element of the set. In this way the postcondition for the iterate set node is $\forall x \in o \mid R$.

For the loop iterate multivalue, let I_m be a iterate multivalue node such as $I_m : \langle i, values \rangle \rightarrow \langle o \rangle$. The precondition for iterate multivalue is defined as the conjunction between the precondition inferred for i and for $values$. The precondition for i is defined as the postcondition for the correspondent input parameter node in the parametric subgraph. Let T_1 be this precondition. In the case of the *values* input, the correspondent input parameter node only refer to one element of this set, let T_2 be its postcondition. The precondition for the entire set is $\forall x \in values \mid T_2$. In this way the precondition for the iterate multivalue node is $T_1 \wedge \forall x \in values \mid T_2$. The postcondition is defined directly from the output parameter node.

5.3 SEMANTIC COMPOSABILITY

Semantic Composability addresses the meaning of the data and the computational processes and aims to verify whether the combination of processes can derive a meaningful result or even one that is the intended result of the user. It is built on the concept of *semantic interoperability*, the ability of independently defined services to exchange information in an unambiguous way, which allows a "shared understanding" of data and functionality. To verify Semantic Composability, and for services to be considered semantically interoperable, it is needed that all the components in a service composition are semantically enabled. A service is considered *semantically enabled* if it has semantic information associated with the provided data and metadata.

Semantic Composability can be static or dynamic. We define different categories of Static Semantic Composability based on which information of the service is semantically enabled: Input/Output (I/O), Attribute, Service.

The *I/O Semantically Composable* category uses semantic information associated with inputs and outputs of services to prevent unexpected behavior of processes at the thematic level. For example, suppose we operate a process *floodRisk*, which calculates the risk of flood in a particular area based on the geometry of the river (`linestring`), the soil types, the digital surface model (DSM), and rainfall data. A client can provide road data as an input instead of river data, in such a way that it also satisfies the Static and Dynamic Syntactic Composability. The process will run without errors. However, the result will not be meaningful, i.e., it will be semantically invalid. To prevent such cases, we complement inputs and outputs with semantic information. This information can be materialized with the use of an ontology, a formal description of a conceptualization [89], specifying concepts and relationships. Languages such as Web Ontology Language (OWL) [90] and Resource Description Framework (RDF) [91] allow the organization of knowledge in hierarchical classes and define relationships between them. Examples relationships are *equivalentClass*, where two classes represent the same concept, and *subClassOf*, where one class inherits the characteristics of its superclass.

RDF expresses statements in a triple form, consisting of a subject, a predicate, and an object. For example, we can state that the *Rhine* belongs to the class *River*, this relation is expressed in Turtle syntax [92] as *Rhine rdfs:type River* where *rdfs* is an RDF vocabulary. An *RDF vocabulary* is used to uniquely identify the context of classes and relationships. It is a collection of Internationalized Resource Identifiers (IRI) with a corresponding namespace prefix. Examples vocabularies are *rdf*, *rdfs*, *owl dbpedia*, *ogc*, *schema*, and *agrovoc*. The notation *dbpedia:River* states that the class *River* belongs to the vocabulary dbpedia, and, as the IRI for the *dbpedia* vocabulary is "http://dbpedia.org/ontology/", information about the class *River* can be found at "http://dbpedia.org/ontology/River".

If the orchestration engine cannot match the semantics of the input and output (by equivalentClass or subClassOf), the services are considered not to be composable.

Example 5.10. Consider the processing service $floodRisk : \langle \text{river}, \text{soil}, \text{dsm}, \text{rainfall} \rangle \rightarrow \langle \text{risk} \rangle$. Semantics are added to the inputs by the use of OWL in the following way:

river owl:equivalentClass dbpedia:River
soil owl:equivalentClass dbpedia:Soil
dsm owl:equivalentClass ogc:DSM
rainfall owl:equivalentClass agrovoc:Rainfall

Suppose we have an input with owl:equivalentClass BraidedRiver, such that:

BraidedRiver rdfs:subClassOf dbpedia:River,

since BraidedRiver is a subclass of River it can be used instead of it, in this way it is a valid input. Analogously, suppose an input with owl:equivalentClass schema:RiverBodyOfWater, such that:

schema:RiverBodyOfWater owl:equivalentClass dbpedia:River

since the classes is equivalent, it is also a valid input.

⌟

The *Attribute Semantically Composable* category in Semantic Composability associates semantics to attributes of inputs and outputs. Observe that for previous levels of composability, we did not define any operator to rename attributes. In practice, when composing services, one may use attributes that represent the same concept under different names, due to language differences or institutional conventions. For example, one can define the attribute that represents the geometry of the feature as *geom*, *geometry* or *geometria* (Portuguese). This is obviously very common in data exchange applications, and it is an unreasonable expectation to work under a single convention for each (semantic) attribute name.

To allow flexibility in the attribute names while maintaining identifiable meanings, the attributes can be linked to an ontology. One can see attributes of a feature type as a relationship in an ontology, which provides a formal description method to identify the meaning of that attribute. Semantic Composability performs an alignment of ontologies that describes the attributes of features, thereby standardizing the attribute names across the composition. This allows verification of the relationship *equivalentProperty* across the defined service descriptions, and a collection of ontologies known to the verification engine, and thus allowing to normalize the names of attributes that represent the same concept.

This alignment of attribute names is performed immediately after the Structural Composability verification, enabling verification of service types in the Static Syntactic Composability and correct association of variables in Dynamic Syntactic Composability. After name normalization, if the orchestration engine cannot match the attribute names, the services are considered to be not composable, since one cannot verify the attributes in Static and Dynamic Syntactic Composability. To the semantic alignment be scalable ideally it should not be embed in the processing service, which turns the WPS complex by adding a reasoner, and adds an overhead by requiring every process to start with a semantic alignment. The client should be able to verify the processing service metadata, identifying which ontology it needs to align and send the data in the schema that the service expects. Additionally, to perform this alignment most likely, it will need the assistance of a Web Ontology Service (an ontology catalogue). This communication with the catalogue would add another overhead to the processing service

Example 5.11. Let *getRivers* and *simplifyLines* be services such that Q_1 is the *getRivers* postcondition and P_2 is the *simplifyLines* precondition. The service types, preconditions, postconditions, and attribute semantics are:

$$\text{getRivers} : \langle\,\rangle \to \langle\ \text{rvr} : \mathbb{P}\langle \text{name} : \texttt{string}, \text{length} : \texttt{real}, \ \text{the\_geom} : \texttt{linestring}\rangle\ \rangle$$

$$\begin{aligned} Q_1 \equiv\ & rvr.the\_geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & rvr.the\_geom{\uparrow}bbox = [5.12, 52.77, 5.53, 53.24] \wedge \\ & rvr.the\_geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\ & rvr.the\_geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & rvr.the\_geom{\uparrow}dimension = [\text{“}x''\text{, “}y''] \end{aligned}$$

name owl:equivalentProperty schema:name
length owl:equivalentProperty dbpedia:length
the_geom owl:equivalentProperty ogc:geometry

$$\text{simplifyLines} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{linestring}\rangle,\ \text{tolerance} : \texttt{real}\ \rangle \rightarrow \langle\ \text{simplified} : \mathbf{typeof}(\ ftr\ )\ \rangle$$

$$P_2 \equiv \quad ftr.geom{\uparrow}dimension = [\text{“}x''\text{, “}y''] \wedge tolerance > 0$$

geom owl:equivalentProperty ogc:geometry

In the composition getRivers; simplifyLine($ftr \leftarrow rvr, tolerance \leftarrow 0.5$), the precondition cannot be evaluated, since the attribute names do not match. Also, type checking fails, due to attribute name differences. Based on the semantic information, however, we can infer that the_geom owl:equivalentProperty geom, and can rename the attributes accordingly.

⌟

Remark. One should note that this evaluation is simple and does not require the use of an inference engine. It only requires a graph traversal in edges of type equivalentProperty. This solution is just applicable to a limited number of cases since, in the case of cross-community interoperability, the concepts will most likely not match directly.

The next is the *Service Semantically Composable* category. It provides semantic information about the overall functionality of a processing service. This category intends to study and allow service substitution. Substitutability is classified as single service and multi-service modes. *Single service substitutability* addresses the case that a particular service is not available during the composition execution, and equips the orchestration engine to find an equivalent service with the same functionality. *Multi-service substitutability* addresses the case that an equivalent service was not found, and multiple parts of the composition need to be added or redesigned to achieve overall equivalent functionality.

To these ends, we need to link an ontology with the service based on the classification of its functionality. Brauner [93] provides a useful general classification of geoprocessing functionality for discovery in catalogue services. This classification provides semantics for a service category. However, the work requires further extension into a full-fledged ontology. Services that are in an equivalentClass or subClassOf relation are potential candidates for single service substitutability.

Apart from semantic comparisons at service category level, one needs to compare services according to Static Syntactic Composability, applying the service subtyping mechanism discussed in Chapter 4, in Dynamic Syntactic Composability, applying the rules of consequence mechanism described in Section 5.2.1, and finally, one must have semantic equivalence in attributes and I/O categories.

Multi-service substitutability is not discussed in this thesis, but certainly deserves later work. The main concept that is required for multi-service substitutability is how a particular process affects the semantics of its inputs, meaning that we need a mechanism that derives the semantics of the output based on the semantics of the input and that of the operation. Once this is possible, the orchestration engine can search for service combinations that provide the needed semantics.

Remark. The concept of substitutability can also be used in Qualitative Composability, where an orchestration engine can substitute services to achieve better quality in the results, to complete the process at better performance, and others. Also, Multi-service substitutability can be used for

composition optimization, identifying parts of the workflow that generate equivalent data, and that thus prevent the unnecessary execution of composition parts when that result is available.

For Dynamic Semantic Composability we will consider the semantics of a collection of services abstracted as a subgraph. To specify the semantics of a workflow, we need the idea of semantic propagation. It can be defined as the dynamic generation of the semantics of an output based on the semantics of the inputs and the semantics of the executed process. Since the output of most atomic operations does not have a specific meaning, the output semantics are better expressed by an algebra that manipulates semantics. For example, if one applies a buffer operation to a road feature, the output does not have any special meaning. However, we can express it for example as *swrlb:Buffer(roads, 50)*. The notation used was Semantic Web Rule Language (SWRL) [94], a combination of OWL and RuleML, which allows a higher-level of abstraction for semantics. In SWRL, the idea of Built-Ins is presented, which is a way of expressing semantics based on a function application.

Every processing service semantic introduced in the service level is considered a built-in and the semantics of the generated output are expressed in terms of the built-ins and the semantics of their inputs. One can chain such a process to express the semantics of a workflow. This idea was not implemented in this thesis, and will require substantial study to allow standardized functional semantics denotations.

Semantic Composability is also needed for automatic service composition. In this case, the client expresses his requirements for the workflow, and the orchestration engine tries to find a service or a composition that fulfills the client's requirements. This concept is not explored in this thesis. The main idea needed to achieve such automatic service composition is a notation to specify the client requirements, which ideally should be generated automatically from a natural language description. An orchestration engine capable of verifying the four described categories of semantic composability could create multiple possible workflows based on the given description. Another challenge is to keep the search space under control when creating possible compositions that fulfill the requirements.

5.4 SUMMARY

In this chapter, we discussed the last two levels of composability considered in this thesis. *Dynamic Syntactic Composability* uses an extension of Hoare Logic that relates the inputs and outputs of a service with its precondition and postcondition. We defined the most common variables used in geospatial services, as well as five operations that allow the manipulation of precondition and postcondition, based on the interaction between the inputs and outputs of the web services.

Semantic Composability addresses the meaning of the data and processes and tries to capture the intended goal of the user. We discussed Semantic Composability in four different categories: I/O, Attribute, Service, and Workflow. We provided a mechanism to normalize attribute names, which removes the need for an attribute renaming method at all other levels of composability. We also proposed a mechanism for service substitutability, based on a service classification by a geo-operator thesaurus.

Chapter 6 discusses how the concepts defined in Chapters 4, 5, and 6 are serialized as JSON and in Chapter 7 we discuss the implementation of the verification. Also, in Chapter 6, we discuss how OGC services can be semantically enabled and how compositions can be shared in an interoperable way.

Chapter 6
Semantic Descriptions

6.1 INTRODUCTION

In this chapter, we present solutions to four different points of attention related to descriptions of OGC services:

- the need for semantically enabled data and metadata.
- flexible machine-readable service functionality description.
- dynamic generation of metadata for WPS results.
- interoperable sharing of compositions.

The first issue is that OGC services are syntactically but not semantically interoperable. There is a series of challenges in making OGC services semantically interoperable, requiring the extension of both metadata and data. Chapter 2 presents GeoJSON as the format for sharing geospatial data in the proposed RESTful OGC services. However, this format lacks semantic information and presents a series of incompatibilities with OGC services.

OGC standards are used in many domains which make predicting user requirements essentially impossible. We thus need flexible and expansible metadata, which allows service functionality to be expressed in any desired granularity. Ideally, such metadata should allow existing services to extend their descriptions to meet OGC requirements. Those descriptions should allow an autonomous client to use OGC web services without being specially programmed against each service. In this thesis, those autonomous clients are called *generic clients*: clients that at run-time can discover the service functionality without relying on human-readable documentation or having standardized URLs.

Chapter 2 proposes service result of asynchronous WPS execution to be indistinguishable from a WFS resource. For both resource types to be truly equivalent, they need to provide equivalent metadata. Currently, WPS result metadata is static and does not depend on the inputs of the process. Metadata should be dynamically generated for the results of an asynchronous WPS execution, based on the inputs' metadata and chaining rules provided by the WPS. The last point of attention is how to share service compositions in an interoperable way. At present, orchestration of OGC services is hard-coded in a WPS, or is described by the use of BPEL and WSDL. The latter makes the composition process complex and requires a specialized tool chain.

In this chapter, we discuss how semantic descriptions can be defined and generated for geospatial data, service metadata, and compositions. Section 6.2 addresses the use of JSON-LD to enhance data and metadata with semantic descriptions. Also, we look into extending GeoJSON to be in accordance with the WFS specification, and to be semantically enabled. Section 6.3 presents the Hydra Core Vocabulary and describes integration with OWS metadata that provides machine-readable and machine-interpretable service descriptions at a level of granularity that matches developer needs. Section 6.4 presents a mechanism of metadata propagation, which allows dynamic

generation of metadata for WPS results, and consequently for compositions. Lastly, Section 6.5 presents a JSON Workflow notation to describe the web service composition in an interoperable way.

6.2 SEMANTIC ENABLEMENT

OGC standards have made significant progress towards syntactic interoperability of services and feature-level data access. However, they do not yet solve semantic heterogeneity problems [9]. In this section, we discuss how OGC services can become semantically interoperable, meaning that services can exchange information in an unambiguous way, with appropriate (deeper) understanding of the meaning of the data [10]. Semantic interoperability allows the use of data in contexts for which it was not originally created and for use by autonomous clients. These can then, for example, reason over multiple service descriptions and assist a user in finding best-fitting services given user requirements.

To achieve semantic interoperability, alternatives in Semantic Web Technology have been proposed [95, 96]. The Semantic Web is an extension of the Web based on W3C standards that facilitate data sharing and re-use across applications by providing accurate and unambiguous definitions with data schemas. One way of achieving the vision of the Semantic Web is Linked Data, a group of best practices to publish and interlink structured machine-readable data on the web [11]. In summary, such best practices include standard identification of resources (URI); use of the Resource Description Framework (RDF) graph-based data model to structure and interlink data; and use of the SPARQL Protocol and RDF Query Language (SPARQL), the standard query language for RDF. One usually combines RDF with an *ontology*, a formal description of a conceptualization [89], specifying concepts and relationships. RDF data combined with ontologies is highly interoperable, being independent of syntax, and can provide unambiguous meaning for the objects and properties.

The lack of semantically annotated services is considered one of the greatest problems to achieve cross-community interoperability [97], and several works state the importance of semantically enabling geospatial data and metadata [98, 99]. The report OGC Testbed 11 [97], recommends that the semantic enablement of OGC services should focus on a RESTful implementation, and the design proposed in this thesis matches that wish.

6.2.1 Previous Approaches

A series of works have attempted to adapt OGC services to provide semantically enabled data. Zhang et al. [100] and Zhao et al. [101] use a conversion from SPARQL queries to WFS filter encoding to query multiple WFS services at once from a single SPARQL endpoint. This solution allows the semantic search of features, however, the data provided is not semantically enabled, and the user needs to provide an appropriate SPARQL query. Conversely, Jones et al. [102] propose the LOD4WFS Adapter, which converts WFS requests into SPARQL queries and translates the RDF response into GML. This solution allows the WFS standard interface to provide Linked Data. However, it does not improve in any respect the semantic issues of both data and metadata of a WFS.

Janowicz et al. [103] propose two more types of services: Web Ontology Services (WOS) and Web Reasoning Services (WRS). An existing WFS service is annotated by providing a link between its data and the ontologies provided by the WOS. This solution potentially provides semantically enabled data and descriptions, as well as semantic search and reasoning capabilities. However, it follows the current implementation of OGC specifications and ignores current trends in web technology. Therefore, a rather complex implementation can only arise.

6.2.2 JSON-LD

To maintain consistency, ease of use, and alignment with contemporary technology, we desire a JSON-based format capable of expressing semantics. We found five attempts of adding hypermedia support to JSON files: JSON Reference [104], HAL [105], Collection+JSON [106], SEREDASj [107], and JSON-LD [108]. The first three address new media type formats are requiring specific processing methods. SEREDASj takes a closer approach to plain JSON and allows to transform the representation into RDF, but this project was eventually discontinued in favour of JSON-LD. JSON-LD is a W3C recommendation and allows the serialization of Linked Data in JSON. JSON-LD builds up from JSON and JSON-LD files are valid JSON files. Its simplicity and direct compatibility with the JSON format make this format a good candidate for the proposed services. Also, a Testbed 11 report [97] recommends that if the service is JSON-based and RESTful, one should add semantics by using JSON-LD.

The only overhead created in JSON-LD is the addition of the *@context* property, which links the attributes with the correspondent vocabulary. This context allows a mechanism similar to namespace definitions in XML, by assigning each attribute to a dereferenceable URL, which unambiguously identifies that concept. JSON-LD objects allow a lossless roundtrip to the RDF model, which makes the data interoperable while at the same time maintaining the ease to use and transfer to the Web of the JSON format. The only exception of this conversion is data in the shape of a list of lists. In this case, the processing complexity increases, and it was decided that the standard should not allow such structures. We address this limitation in the next section. One should notice that JSON-LD only permits the association of JSON properties with semantics, which means that ontologies are still needed to express them. In [109], OGC proposes a lightweight ontology for some spatial concepts. In this thesis, we use this ontology as the basis for expressing the semantics of spatial concepts. The format also defines a series of attributes, all starting with the "@" character. The three most common attributes are:

- *@context*: a JSON object containing the name of the attributes and the corresponding URL that identifies the attributes semantics.
- *@id*: a string that holds the RDF node identifier. It must be a dereferenceable URL.
- *@type*: a string or array of strings that represents a dereferenceable URL, which identifies the object semantics. Also, the *@type* attribute provides a linking mechanism, being an alternative of XLink to JSON. With these linking capabilities, the proposed REST services comply with the Hypermedia as the Engine of Application State (HATEOAS) constraint that we presented in Chapter 2.

A framing API, which allows restructuring a JSON-LD file into a format more convenient for the user or application, is also developed. Listing 6.1 presents an example of a JSON-LD representation of a river feature. The attribute *@id* provides the URL where one can access the feature, *@type* associates the semantics of the feature to the river ontology of DBpedia. The *@context* relates attributes and the semantics, for example, the attribute *name* is associated with the concept of a name in the Schema.org vocabulary. Lastly, for the attribute *countries* the *@type* attribute (line 15) is used to denote that each element is a link, in a similar way to XLink.

Listing 6.1: JSON-LD example

```
1 {
2 @context: {
3   name: "http://schema.org/name",
4   xsd: "http://www.w3.org/2001/XMLSchema#",
5   uom: "http://www.opengis.net/def/uom",
```

```
  length: {
   @id: "http://schema.org/distance",
   @type": "xsd:float",
   uom: "http://www.opengis.net/ont/measure/spatial/uom#Kilometer"
  },
  dbpedia: "http://dbpedia.org/ontology/",
  dbinstance: "http://dbpedia.org/page/"
  countries: {
   @id: "http://schema.org/nationality",
   @type: "@id"
  }
},
river: {
  @id: "http://www.example.com/wfs/rivers/5",
  @type: "dbpedia:river"
  name: "Amazon River",
  length: 6992,
  countries: ["dbinstance:Brazil", "dbinstance:Colombia", "dbinstance:Peru", "
      dbinstance:Ecuador"]
}}
```

We use JSON-LD as the main format for the proposed service for data and metadata sharing. By combining it with ontologies, we turn OGC services into semantically enabled services. In the next section, we define a GeoJSON extension based on JSON-LD, which enables spatial data to be semantically enabled. Section 6.3 presents the Hydra Core Vocabulary, which uses JSON-LD to provide semantic metadata to a service, describing its functionality.

6.2.3 GeoJSON Extension

Chapter 2 introduces GeoJSON as the preferred JSON format for serializing geospatial features, and we present a series of disadvantages in its use. In this section, we introduce *ExtGeoJSON*, an extension of GeoJSON that allows the representation of geospatial objects consistent with JSON-LD, and GeoJSON.

The first problem that we address is how to add semantics to GeoJSON objects. To this end, GeoJSON can be extended with JSON-LD principles. An important problem of using GeoJSON as such is that one cannot efficiently encode a list of lists in RDF [110], and that this requires a modification in GeoJSON or the JSON-LD parser. Gillies [111] discussed the GeoJSON-LD standard, however, consensus on approach did not emerge. In this thesis, we propose the encoding of the geometry in RDF in the WKT format. Since it is a string, the conversion does not create any issues. Also, WKT is the most common used format for expressing geospatial features in Linked Data, being part of the GeoSPARQL standard. One solution to achieve this without breaking compatibility with both JSON-LD and GeoJSON is to create a custom type in the JSON-LD standard that treats the value of the property as a string, essentially ignoring any complex parsing. In this thesis we identify this JSON-LD type as *jsonld:geoJsonCoords*. A client that wants to make use of the spatial coordinates in the RDF representation can use a custom parser that converts objects of type *jsonld:geoJsonCoords* to WKT. As discussed in the previous section, to add semantics to features, one can use the JSON-LD property *@type*.

The second problem is how to handle coordinate systems in GeoJSON. The latest draft from GeoJSON advocates only the use of EPSG 4326, stating that allowing multiple coordinate systems can disrupt interoperability. However, this statement goes against general experience and requirements from OGC standards. Therefore, in OGC service context GeoJSON should be able to express geometries in any given coordinate system. The property *crs* has the context *"http://www.opengis.net/ont/spatial/crs"* with possible values following the URL convention of OGC Name Type Specification for Coordinate Reference Systems [83]. For example: *"http://www.opengis.net/def/crs/OGC/1.3/CRS84"*.

The third problem is how to handle multiple measurement values in GeoJSON. Currently, the specification does not mention measurement values. However, there is no restriction in the dimension of the geometry, only restricting it to numeric values. GeoJSON-X [112] is a GeoJSON extension that allows the representation of multiple measurement values. Those values can have numeric or string types. The property *extension* was added to the *geometries* object, which allows one to specify the identifiers of the measurement values. This approach is not valid GeoJSON according to the specification. However, it is compatible with current Leaflet and OpenLayers implementation. We incorporate this approach and extend it to JSON-LD. We propose the following context for the extension object *"http://www.opengis.net/ont/spatial/measure"*. It is of type array of strings, where the strings provide dereferenceable identifiers. Table 6.1 exemplifies measurement values used in GPS Exchange Format (GPX) and TrackPointExtension and extension of GPX defined by Garmin. The table was extracted from [112]. Listing 6.2 presents an example of an ExtGeoJSON expression that represents a river, with a measurement value that provides the depth, with an explicit coordinate system, and an *"@id"* indicating the location in the WFS where it was originally accessed.

Table 6.1: Examples of measurement values

Value	Description	Source
atemp	air temperature in degrees Celsius	TrackPointExtension V1
cad	cadence in revolutions per minute	TrackPointExtension V1
depth	diving depth in meters	TrackPointExtension V1
hr	heart rate in beats per minute	TrackPointExtension V1
time	time stamp (defined as xsd:dateTime in GPX 1.1) for an individual geographic poin.	GPX 1.1
wtemp	water temperature in degrees Celsius	TrackPointExtension V1

Listing 6.2: ExtGeoJSON example

```
{@context: {
  type: {
    @id: "http://www.opengis.net/ont/geojson/type",
    @type: "@id"
  },
  geometry: "http://www.opengis.net/ont/geojson/geometry",
  coordinates: {
    @id: "http://www.opengis.net/ont/geojson/coordinates",
    @type: "jsonld:geoJsonCoords"
  },
  dbpedia: "http://dbpedia.org/ontology/",
  line: "http://www.opengis.net/ont/sf#LineString",
  feature: "http://www.opengis.net/ont/geojson/feature",
  extensions: {
    @id: "http://www.opengis.net/ont/spatial/measure",
    @type: "@id"
  },
  crs: "http://www.opengis.net/ont/spatial/crs",
  depth: "http://www.opengis.net/ont/spatial/measure/depth"
},
@id: "http://www.example.com/wfs/rivers/5"
type: "feature",
@type: "dbpedia:river",
crs: "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
geometry: {
  type: "line",
  extensions: ["depth"],
  coordinates: [[102.0, 0.0, 0.5], [103.0, 1.0, 0.8], [104.0, 0.0, 0.7], [105.0, 1.0,
      0.4]]
```

```
29 },
30 properties: {
31     name: "Igarape"
32 }}
```

The last problem is how to obtain efficient GeoJSON data transfer. The GeoJSON format is not as efficient as binary encodings such as Shapefile or WKB. Nevertheless, GeoJSON is still more efficient than GML 2.0 for simple data and is as efficient for complex data [113]. Several approaches can be used to reduce the size of GeoJSON files. The first is to minify the GeoJSON file. Libraries such as GeoJSON-minifier [114] perform a precision reduction and uses delta and zigzag encoding to reduce the storage size of the coordinates. In general, it reduces the file size by a factor of two to four. The obvious disadvantage is that it creates overhead on both server and client to serialize and deserialize the object.

A relevant second approach is by compressing the GeoJSON file. The HTTP specification recommends the use of *Gzip* to compress data transferred over the Web. Another compression mechanism is GeoBuf. *GeoBuf* [115] is a lossless compression of GeoJSON data into a binary format. It claims to make GeoJSON files six to eight times smaller, has fast en- and decoding, and could potentially allow incremental reading of features. It is based on Google Protocol Buffers, a language-neutral mechanism for serializing structured data into a compact binary wire format. When compared, GeoBuf provides better compression than Gzip and also gives faster en- and decoding. The data compression can assist in the reduction of bandwidth use in a service composition.

6.3 HYDRA CORE VOCABULARY

The literature presents an extensive list of service description languages. In this section, we discuss the most relevant ones for use in OGC services and RESTful APIs. WSDL [116] is an XML-based description of SOAP services. In its version 2.0, it was extended to describe RESTful services. It can describe both REST and SOAP services, however, REST services should be described from an RPC perspective. OWL-S [117] and WSMO [118] were two important initiatives in the area of semantic descriptions for Web Services. Both predate the idea of Linked Data, and we are not aware of any substantial real-world usage of these languages [119]. hRESTS [120] is a microformat approach, meaning that the semantics are associated with HTML, not applying to OGC services. Also, the descriptions are based on RPC semantics, not RESTful bindings. SA-REST [121] is similar to hREST, however, it uses RDFa syntax for service descriptions, which is an extension of HTML. SEREDASj [107] was created to address the problems of hRESTS and SA-REST, allowing a RESTful description based on resources and HTTP-like semantics for interactions. It uses JSON-like format for the description. The work in SEREDASj was discontinued in favour of the work in JSON-LD [53].

Lanthaler, 2013 [122] presents the Hydra Core Vocabulary, which allows to model and describes RESTful services using JSON-LD and a specific vocabulary. The goal is to enable autonomous clients to interact with the service. Hydra descriptions allow a client to discover the functionality at run-time, removing the need of standardized URLs. In this way, a Hydra-enabled service is truly self-described by the use of hypermedia controls, lowering the need for human-readable documentation. Since Hydra is used to describe the API interface from Linked Data Fragment datasets [123], more than 600,000 APIs are described by Hydra Core Vocabulary [124].

In this thesis, we adopt the Hydra Core Vocabulary to describe the metadata of services. Since it uses JSON-LD, we maintain the consistency of formats in the services. Also, the use of Hydra Core Vocabulary allows easier communication between OGC services and RESTful web services

not in the OWS-family. We combine this vocabulary with standard OGC metadata, and a JSON serialization of the concepts presented in Chapters 3, 4, and 5, which allows the services to have enough metadata for the composition verification at compile-time.

Hydra adds two classes to the *GetCapabilities* document: *supportedClass*, a list of resources available for the service, and *supportedStatus*, the description of the HTTP status code that the client is expected to handle. The resource description contains information about the *supportedOperations*, descriptions about the HTTP operations that one can perform on the resource, and *supportedProperties*, the attributes that the resource supports. Also, the vocabulary completely describes all the functionality of the service, allowing a client to discover it at runtime without the need of external documentation. Also, by the use of template links, URL query parameters can be documented. In this way, different operations and query parameters for each resource can be documented. The complete specification can be accessed at [125]. Additionally to the Hydra properties, we have added custom metadata related to precondition, postcondition, service types, and semantics.

Example 6.1. Consider the service *computeVoronoi* with the following description:

$$\text{computeVoronoi} : \langle\ \text{ftr} : \mathbb{P}\langle \text{geom} : \texttt{point}\rangle,\ \text{bb} : \texttt{bbox}\ \rangle \\ \rightarrow \langle\ \text{voronoi} : \mathbf{typeof}(\ ftr\ ) \oplus \langle \text{geom} : \texttt{polygon}\rangle\ \rangle$$

$$\begin{aligned} P \equiv\ & \text{within}(ftr.geom{\uparrow}bbox, bb) \wedge \\ & bb{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & ftr.geom{\uparrow}dimension = [\text{“x”},\text{“y”}] \wedge \\ & bb{\uparrow}format = \text{“GeoJSON”} \\ Q \equiv\ & voronoi.geom{\uparrow}crs = ftr.geom{\uparrow}crs \wedge \\ & voronoi.geom{\uparrow}bbox = bb \wedge \\ & voronoi.geom{\uparrow}dimension = [\text{“x”},\text{“y”}] \wedge \\ & voronoi.geom{\uparrow}bboxCrs = ftr.geom{\uparrow}crs \wedge \\ & voronoi.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \end{aligned}$$

Listings 6.3 and 6.4 exemplify the serialization of the description in JSON. The attributes *required* and *unique* (lines 4 and 5 of Listing 6.3) refer to the conditions added to the input node in Chapter 3, which specify whether the input is required for the execution of the process, and whether multiple edges can connect to the input.

Listing 6.3: JSON representation for service type

```
1  inputTypes: {
2   bb: {
3    type: "BBOX",
4    required: true,
5    unique: true
6   },
7   ftr: {
8    type: {$set: {$record: {geom: "Point"}}},
9    required: true,
10   unique: true
11  }
12 },
13 outputTypes: {
14  voronoi: {
15   typePropagation: {$addAttrs: [
16    {$typeOf: "features"},
17    {geom: "Polygon"}
18   ]},
```

```
  type: {$set: {$record: {geom: "Polygon"}}}
 }
}
```

Listing 6.4: JSON representation for pre- and postcondition

```
precondition: {
 $and: [
  {$eq: ["bb.crs","ftr.geom.crs"]},
  {$eq: ["ftr.geom.dimension",{$literal: ["x","y"]}]},
  {$eq: ["bb.format",{$literal: "GeoJSON"}]},
  {$within: ["ftr.geom.bbox", "bb"]}
 ]
},
postcondition: {
 $and: [
  {$eq: ["voronoi.geom.dimension", {$literal: ["x","y"]]},
  {$eq: ["voronoi.geom.crs","ftr.geom.crs"]},
  {$eq: ["voronoi.geom.bbox","bb"]},
  {$eq: ["voronoi.geom.bboxFormat",{$literal: "GeoJSON"}]},
  {$eq: ["voronoi.geom.bboxCrs","ftr.geom.crs"]}
]
}
```

Appendix C presents the JSON serialization for types, preconditions, and postconditions. Chapter 7 describes a generic client capable of processing Hydra descriptions for OGC services, which allow the dynamic creation of a user interface on the basis of the available service functionality.

6.4 METADATA PROPAGATION

In Chapter 2, we modified the WPS standard to ensure that the result of an asynchronous process is indistinguishable from that of a WFS. For this to happen, it is needed that the metadata of the result contains all the required metadata of a WFS resource. In the current WPS specification, the metadata for the results is general and does not depend on the inputs of the process. We propose a metadata propagation mechanism, which allows the dynamic metadata generation of WPS results. This mechanism directly relates to the methods presented in Chapters 4 and 5.

We divide this mechanism into three parts: generation of JSON-LD Context, generation of JSON Schema, and generation of general metadata. For the generation of JSON-LD context, the WPS uses its service signature to modify the *@context* object from its inputs. This information can be transferred with the input data or can be requested by the *DescribeFeature* operation discussed in Chapter 2. We use the type propagation operators described in Chapter 4 to modify the JSON Context of the inputs. The *typeof* operator indicates which inputs must provide the *@context* object for metadata propagation. In case that an attribute is added to the output by the *addAttrs* operator, the WPS service is required to provide the metadata for that attribute.

Example 6.2. Consider the service *aggregate* which receives two inputs: *ref*, a feature collection with geometry `polygon` or `multipolygon`, and *pnts* a feature collection with geometry of type `point`. The service has one output, *agg*, which returns the input *ref* with an added attribute *pntCount*, which is the number of points inside each geometry. The service type and postcondition are the following:

$$\text{aggregate} : \langle\ \text{ref} : \mathbb{P}\langle \text{geom} : \text{union}(\ \texttt{polygon}, \texttt{multipolygon}\ )\rangle,\ \text{pnts} : \mathbb{P}\langle \text{geom} : \texttt{point}\rangle\ \rangle \\ \rightarrow \langle\ \text{agg} : \textbf{typeof}(\ \textit{ref}\ ) \oplus \langle \text{pntCount} : \texttt{integer}\rangle\ \rangle$$

$$\begin{aligned} Q_1 \equiv\ & agg.geom{\uparrow}crs = ref.geom{\uparrow}crs \wedge \\ & agg.geom{\uparrow}bbox = ref.geom{\uparrow}bbox \wedge \\ & agg.geom{\uparrow}dimension = [\text{“x”},\text{“y”}] \wedge \\ & agg.geom{\uparrow}bboxCrs = ref.geom{\uparrow}bboxCrs \wedge \\ & agg.geom{\uparrow}bboxFormat = ref.geom{\uparrow}bboxFormat \wedge \\ & agg.pntCount \geqslant 0 \end{aligned}$$

$$\text{pntCount} : \textit{“http://www.opengis.net/ont/measure/quantity\#CountingUnit”}$$

The *typeof* operator requires the context of the materialized input for *ref*, and the *addAttrs* operator indicates that a new property *pntCount* is added to that context. Consider the following service as input for *ref*:

$$\text{neighborhoods} : \langle\ \rangle \rightarrow \langle\ \text{nbhd} : \mathbb{P}\langle \text{geom} : \texttt{polygon},\ \text{name} : \texttt{string},\ \text{population} : \texttt{integer}\rangle\ \rangle$$

$$\begin{aligned} Q_2 \equiv\ & nbhd.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & nbhd.geom{\uparrow}bbox = [3.3579, 50.7504, 7.2275, 53.5552] \wedge \\ & nbhd.geom{\uparrow}dimension = [\text{“x”},\text{“y”}] \wedge \\ & nbhd.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & nbhd.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \end{aligned}$$

```
{@context: {
  type: {
    @id: "http://www.opengis.net/ont/geojson/type",
    @type: "@id"
  },
  geom: "http://www.opengis.net/ont/geojson/geometry",
  coordinates: {
    @id: "http://www.opengis.net/ont/geojson/coordinates",
    @type: "jsonld:geoJsonCoords"
  },
  dbpedia: "http://dbpedia.org/ontology/",
  polygon: "http://www.opengis.net/ont/sf#Polygon",
  feature: "http://www.opengis.net/ont/geojson/feature",
  name: "http://schema.org/name",
  population: "http://dbpedia.org/ontology/populationTotal"
}}
```

The propagation mechanism will simply add the *pntCount* property to the *@context* object and in this way it can accurately describe the feature schema.

⌟

Remark. In this thesis, we do not take into account semantic propagation as discussed in Chapter 5. In a system that realizes this, the output semantics will change, and this implies a change in the *@type* property, which may require modification in the *@context*.

The generation of the JSON Schema works analogously. The WPS service requests the JSON-Schema of the relevant inputs, and based on the type propagation operators modifies the schema. For the general metadata, we cover two cases: output type and postcondition. The output type is generated by the mechanism of type propagation described in Chapter 4. When the inputs are materialized, the output type can be evaluated. The WPS service must request metadata information about the types of the materialized inputs. In the case of the example, the propagated output type is:

$$\mathbb{P}\langle \text{geom} : \texttt{polygon},\ \text{name} : \texttt{string},\ \text{population} : \texttt{integer},\ \text{pntCount} : \texttt{integer}\rangle$$

The postcondition can be evaluated using the postconditions of the inputs (which need to be requested), and the rules presented in Chapter 5. In the example case, the propagated postcondition is:

$$\begin{aligned} Q_1 \equiv\ & agg.geom{\uparrow}crs = \text{“http://www.opengis.net/def/crs/EPSG/0/28992”} \wedge \\ & agg.geom{\uparrow}bbox = [3.3579, 50.7504, 7.2275, 53.5552] \wedge \\ & agg.geom{\uparrow}dimension = [\text{“x”},\text{“y”}] \wedge \\ & agg.geom{\uparrow}bboxCrs = \text{“http://www.opengis.net/def/crs/EPSG/0/4326”} \wedge \\ & agg.geom{\uparrow}bboxFormat = \text{“GeoJSON”} \wedge \\ & agg.pntCount \geqslant 0 \end{aligned}$$

The same mechanism can also be applied to derive metadata for a service composition in parametric workflows. Such metadata uses the metadata of the services connected to the input parameter and the output parameter. In this way, the mechanism of metadata propagation can be repeatedly applied, deriving metadata for outputs of each processing service in a composition. The metadata of the service connected to the output parameter is used as the metadata for the workflow. Due to the resource-oriented architecture, the metadata of each output is independent.

It is also important to track the provenance of data generated by a workflow. This was not addressed in this thesis. The Open Provenance Model [126] and the PROV-O [127] ontology are possible candidates to be followed.

6.5 JSON-W

As OGC standards are based on the concepts of services and data bindings, they do not provide any means of abstracting compositions. In this thesis, a composition is desired to be indistinguishable from a WPS from a metadata perspective, and to be indistinguishable from WFS from a data access perspective. However, a specific application, an orchestration engine namely, is needed to execute a composition. An *orchestration engine* receives as an input a composition representation, and it handles the execution of each service, the transmission of the necessary data, the exception handling, and it returns the results to the client. This section discusses an interoperable notation that can be used for this purpose, with enough information to execute and verify a composition.

One important concern about workflow representation is interoperability. Ideally, the workflow representation can be used by different orchestration engines and graphical user interfaces. In this thesis, we model the workflow in the Resource Description Framework (RDF) and then serialize this model in the JSON-LD format. The use of RDF and ontologies ensure that one can share workflows in an interoperable manner. By the use of JSON-LD framing, we ensure simplicity of use and sharing on the Web without losing the expressiveness of RDF.

Workflow representations such as Business Process Model and Notation (BPMN) are usually defined in XML and cover an extensive variety of use cases such as conditional flow, message exchange, sub-process, event, and loop. We define a simple workflow notation in JSON-LD inspired in the BPMN notation; it focuses on the definition of connections between inputs and outputs. This simplification allows a concise notation that covers all example cases in this thesis. The proposed language is conceptually different from BPEL, which focuses on message exchange and data bindings. We focus on the modeling aspects, ensuring that the workflow is interoperable, and complete information for the execution or verification of a composition.

Chapter 3 presents a formalization of a composition as a graph. The RDF model is a natural way of representing that graph, in which the triple *subject – predicate – object* can be seen as a relationship between nodes in a graph. In this case, the subject and object are considered nodes of the graph, and the predicates are edges. Aside the RDF representation, we also need an ontology to formalize the classes and properties described in RDF. The ontology can also be used to verify whether a particular workflow in RDF is semantically correct. This model can both describe graphs and parametric graphs (as defined in Chapter 3). Parametric graphs described in RDF model are called *parametric workflows*. Appendix D presents an example of the RDF serialization, ontology, and JSON-LD context.

The RDF model is expressive, however, it is considered too complex to be handled by orchestration engines and applications directly. We make use of JSON-LD framing to do a lossless conversion from RDF to a compact JSON-LD object. By providing a JSON-LD context, one can translate the JSON-LD object back to the RDF graph representation. This notation is compact, not explicitly creating nodes for inputs and outputs.

In JSON-LD, we describe the workflow with two properties: tasks and sequence flows. The property *tasks* is an array containing task objects. A *task* represents a service with service, input, and output nodes. The property *sequence flows* is an array containing sequence flow objects. A *sequence flow* represents the edges of the graph; they define connections between ports (inputs and outputs) of tasks. The task and sequence flow can be at any order; the Orchestration Engine should evaluate the order in which services must be executed. This can be done by applying a topological sorting algorithm on the graph structure [128]. We define nine properties for the task object:

- *id*:`string` a unique identifier within the workflow. This is required since the same service can be used multiple times (for example an intersect WPS can be used in two parts of the workflow). It is of type string.

- *name*:`string` a string which represents the tag of the service node.

- *type*:`string` a string which represents the type of the task. The possible values are: *"WFS"*, *"WPS"*, *"Literal"*, *"Conditional"*, *"Subgraph"*, *"Iterate Input"*, *"Iterate Sets"*, *"Iterate Multivalue"*, *"Input parameter"*, and *"Output parameter"*.

- *inputs*:`array(string)` an array of strings with the tags of the input nodes associated with the service.

- *outputs*:`array(string)` an array of strings with the tags of the output nodes associated with the service.

- *url*:`string` The URL of the service resource. This attribute is of type string, and can only be used by tasks of type *"WFS"*, *"WPS"*, *"Subgraph"*, *"Iterate Input"*, *"Iterate Sets"*, and *"Iterate Multivalue"*. In the case of the last four, the URL refers to the subgraph associated with the loop.

- *value* the value of a literal node, it can be of any valid JSON data type. Only a task of type *"Literal"* can use this attribute.
- *condition*:`record` an JSON object representing a condition statement. Only a task of type *"Conditional"* can use this attribute.
- *reference*:`string` a string representing the name of the input or output linked with the parameter. Only a task of type *"Input parameter"*, or *"Output parameter"* can use this attribute.

We define four properties for the sequence flow object:

- *from*:`string` the id of the task linked to the start point of the edge.
- *to*:`string` the id of the task linked to the end point of the edge.
- *fromPort*:`string` the output node tag linked to the start of the edge.
- *toPort*:`string` the input node tag linked to the end of the edge.

Listing 6.5 presents the JSON-W notation of the workflow for the graph presented in the Figure 6.1.

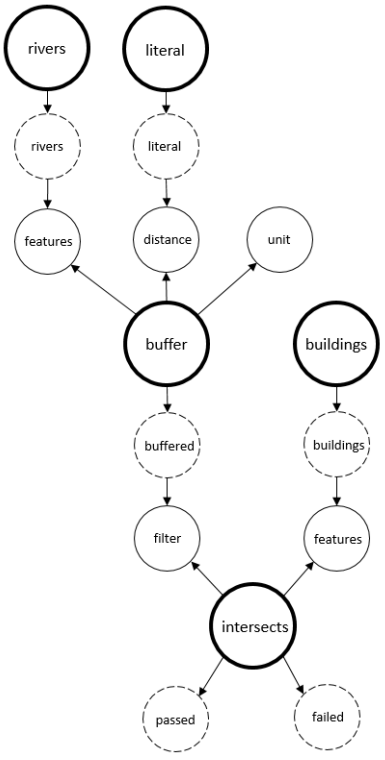


Figure 6.1: Graph of a composition

Listing 6.5: JSON-W example

```
1 {tasks: [
2     {id:"task:0", name: "rivers", type: "WFS", inputs: [], outputs: ["rivers"], url:
          "http://localhost:3001/wfs/rivers/nl.top10nl.111016252"},
3     {id:"task:1", name: "literal", type: "Literal", inputs: [], outputs: ["literal"],
          value: 1000},
```

```
    {id:"task:2", name: "buffer", type: "WPS", inputs: ["features","distance","unit"]
        , outputs: ["buffered"], url: "http://localhost:3000/wps/buffer/jobs"},
    {id:"task:3", name: "buildings", type: "WFS", inputs: [], outputs: ["buildings"],
         url: "http://localhost:3001/wfs/buildings"},
    {id:"task:4", name: "intersects", type: "WPS", inputs: ["filter","features"],
        outputs: ["passed", "failed"], url: "http://localhost:3000/wps/intersects/
        jobs"}
],
sequenceFlows: [
    {from: "task:0", to: "task:2", fromPort: "rivers", toPort: "features"},
    {from: "task:1", to: "task:2", fromPort: "literal", toPort: "distance"},
    {from: "task:3", to: "task:4", fromPort: "buildings", toPort: "features"},
    {from: "task:2", to: "task:4", fromPort: "buffered", toPort: "filter"}
]}
```

Unlike BPEL, the proposed language does not specify the methods for communication and message exchange, as the goal is only to model the composition. This simplification is applicable in this thesis since all the component services are expected to be JSON-LD based RESTful OGC services with Hydra Core Vocabulary metadata. As a recommendation for future work, the notation should be extended to define the transferred data format and the data access bindings of each service explicitly.

We also apply the mechanism of metadata propagation discussed in the previous section to compositions. By using this procedure, the metadata of each component service is propagated to generate metadata of a parametric workflow. Additionally to automatically generate the metadata, the creator of the workflow must specify information about the creation of the workflow, such as name, keywords, and responsible party. This metadata is specified in the same way as in a WPS service, and it allows the description of the datasets and operations used to create the workflow, the provenance of the generated data, workflow discovery, substitutability of services, and semantically aided search of component services during composition creation. In Chapter 7, we describe and implement a Workflow CSW, capable of storing and sharing workflows and parametric workflows encoded in JSON-W.

To enable a more efficient verification of service composability, we include an additional attribute, namely *metadata*. This attribute can be used within a task or in the root of the JSON-W document. In the task object, it captures the service metadata of the resource as described by the Hydra Core Vocabulary. The metadata property in the JSON-W root holds the metadata of the workflow, which allows, for example, workflow discovery. In this thesis, we focus on the metadata of parametric workflows generated by metadata propagation, which allows them to be used as a component in another composition.

Bucchiarone and Gnesi [129] compares the main languages used to describe business process composition, namely BPEL, BPML (deprecated, now BPMN), WSCI, WS-CDL, and DAML-S (superseded by OWL-S). It uses eight criteria to evaluate the languages mentioned above, in which we also evaluate JSON-W:

- Modelling collaboration: the interaction between services is modelled in terms of message exchange. This is a basic function of every workflow language, and it is the basis of JSON-W.
- Modelling execution control: the ability to assemble individual web services in a composition. Also, a basic function of every workflow language.
- Representation of Role: component services play different roles in different execution stages. This requirement relates to choreography, where each service is responsible for handling data communication. JSON-W does not support the representation of role, meaning that its execution needs to be by orchestration.

- Transactions and Compensations: ability to manage transactions (atomic or not) and of rollback effects of completed transactions in case of failure. The proposed language does not support transactions, being left as recommended, future work.
- Exception handling: the language should provide a mechanism to handle cases where a service does not respond or fails in execution. In this thesis, we hand this responsibility to the orchestration engine, simplifying the workflow language to be a concern mostly about modelling aspects.
- Semantic support: languages should enable semantic metadata to facilitate automatic composition and web service verification. In JSON-W, both the workflow and the components can express the semantics of the operations and data. In this thesis, we use this semantic information for automatic verification of the composition at compile-time.

The two remaining aspects are business agreement support and software support, which is not applicable to the proposed language. Wohed at al. [130] evaluates BPEL, XLANG, WSFL, BPML, WSCI considering 20 workflow patterns described in [131]. Table 6.2 was adapted from [130] to include the evaluation for JSON-W. For definitions of each pattern, one can consult the workflow pattern home page [132].

Table 6.2: Evaluation of workflow languages

Pattern	Language					
	BPEL	XLANG	WSFL	BPML	WSCI	JSON-W
Sequence	+	+	+	+	+	+
Parallel Split	+	+	+	+	+	+
Synchronization	+	+	+	+	+	+
Exclusive Choice	+	+	+	+	+	+
Simple Merge	+	+	+	+	+	+
Multi Choice	+	-	+	-	-	+/-
Synchronizing Merge	+	-	+	-	-	+
Multi-Merge	-	-	-	+/-	+/-	-
Discriminator	-	-	-	-	-	-
Arbitrary Cycles	-	-	-	-	-	-
Implicit Termination	+	-	+	+	+	+
MI without Synchronization	+	+	+	+	+	+/-
MI with a priori compile-time knowledge	+	+	+	+	+	+
MI with a priori run-time knowledge	-	-	-	-	-	+
MI without a priori run-time knowledge	-	-	-	-	-	-
Deferred Choice	+	+	-	+	+	-
Interleaved Parallel Routing	+/-	-	-	-	-	-
Milestone	-	-	-	-	-	-
Cancel Activity	+	+	+	+	+	-
Cancel Case	+	+	+	+	+	-

Some of the patterns worth a deeper discussion:

- Exclusive Choice: we use conditional nodes to allow the creation of exclusive Choice patterns.
- Simple merge: there is no need for a specific notation for simple merge since the Orchestration Engine can automatically infer the merge points in a well-formed graph.
- Multi-Choice: JSON-W does not directly support this pattern. However, it can be performed by rewriting the composition using parallel execution and conditionals.
- Synchronizing Merge: the Orchestration Engine can infer the synchronization points in a composition in a similar way to simple merge.
- Multi-merge: it is not supported by JSON-W, in this notation for each service execution one service node must be provided. One possible workaround is the use of iterate input node. However, it is not equivalent to the Multi-merge pattern in the general case.
- Discriminator: this pattern is not supported, however, it is an interesting extension. It allows specifying the case where two or more processes run in parallel and control will consider only the first result to be returned, with others being ignored. This allows the use case of reducing the overall processing time of the workflow by performing an operation in two different service endpoints.
- Arbitrary Cycles: this pattern is intentionally not supported, since it cannot be verified for all cases that the loop will halt.
- MI without synchronization: in JSON-W one can simulate this pattern with the iterate set node. This node necessarily synchronizes the results at the end of the execution. However, one can insert the required operation inside the subgraph associated with the iterate set node, achieving the same functionality.
- MI with a priori compile-time knowledge: in JSON-W we build this pattern with the iterate multivalue node.
- MI with a priori run-time knowledge: we build this pattern with the iterate sets node.
- MI without a priori run-time knowledge: this pattern is intentionally restricted since is not a safe operation, where one is not able to ensure the loop termination.

To extend JSON-W use cases, an appropriate starting point can be the patterns defined in [132]. Also, as a recommendation for future work the proposed language should be compared with other languages that can describe compositions, such as the Sensor Model Language (SensorML) [133] and Open Modelling Interface (OpenMI) [134].

6.6 SUMMARY

This chapter discusses how to extend data and metadata with semantic information, which turns OGC services into semantically enabled ones. We propose the use of JSON-LD to enrich JSON files with semantic capabilities, making metadata of OGC services semantically interoperable. Also, it handles namespace definitions as in XML, can handle links, being an alternative for XLink from XML, and allows the definition of abstract data types by associating the context with an ontology. To provide semantically enabled geospatial data, we introduce ExtGeoJSON, an extension of GeoJSON that is compatible with JSON-LD, therefore giving semantic capabilities to the data.

Additionally, we discuss limitations of GeoJSON such as handling measurement multiple values, geometry semantics, specific feature type as required by the WFS specification, and compaction.

To enhance the metadata of OGC services, we proposed the use of the Hydra Core Vocabulary, which allows the description of service functionality in a machine-readable and machine-interpretable way. Hydra descriptions allow a client to discover the functionality at run-time, without the need to use standardized URLs. In this way, a Hydra-enabled service is truly self-describable by the use of hypermedia controls, lowering the need for human-readable documentation, which allows the creation of generic clients. Also by using Hydra, the OGC Web Services can better integrate with other RESTful APIs.

For the result of an asynchronous WPS execution to be indistinguishable from that of a WFS resource both resources must provide equivalent metadata. We propose a mechanism for metadata propagation that is capable of dynamically generating metadata from a result of a WPS asynchronous process based on the input metadata. This mechanism also allows the metadata of component services in a composition to be combined into the parametric workflow metadata. We focus on the propagation of metadata needed for the verification of the workflow composability.

Lastly, we presented an interoperable workflow representation based on JSON-LD. The developed notation for describing and sharing workflows allows a lossless roundtrip to RDF serialization, making the workflows interoperable and at the same time maintaining the ease to use and transfer to the Web of the JSON format. The aim of this representation is to be used as an input for orchestration engines, and services that can verify the composability of the services.

In Chapter 7, we present the implementation of the proposed RESTful services, of a generic client capable of processing Hydra Core Vocabulary and the dynamical creation of a user interface based on the service functionality. Lastly, we described the implementation of an orchestration client which assists the user in generating the JSON-W notation of a composition.

Chapter 7
Implementation

7.1 INTRODUCTION

In the previous chapters was discussed several modifications to the OGC services that combined can improve the verification and sharing of service compositions. In this chapter, we discuss the implementation of those services, and also present three specialized services. The first is an orchestration WPS, which can leverage the proposed modification to the services, being capable of executing the asynchronous execution mode for WPS services, creating and monitoring jobs according to the specification. Also, the orchestration engine uses a mechanism to pass data by reference, removing the need for the orchestration engine receive data from the services. The second is a composition verification WPS that receives a JSON-W and verifies its validity according to the methods described in Chapters 3, 4 and 5. Lastly, the third is a Workflow CSW, which allows the storage and retrieval of workflows descriptions as described in Chapter 6.

As a proof of concept, two web applications were built first is the Generic OGC Client, which is capable of processing the service metadata described in Chapter 6, which allows seamless interaction with different OGC services. Second is the Orchestration Client, a graphical user interface to assist the construction of services composition by visually creating the workflow representation and by verifying at compile-time the composability of the selected services.

In this chapter is discussed the implementation of the services mentioned above and applications. Section 7.2 discusses the implementation of the services WFS, WPS, and CSW. Section 7.3 describes the implementation of a Generic Client for the implemented services. Section 7.4 discusses the implementation of three specialized services: the Orchestration WPS, the Composition Verification WPS, and the Workflow CSW. Lastly, Section 7.5 describes the implementation of the Orchestration Client.

7.2 SERVICES IMPLEMENTATION

For the service implementation, we use Node.js [135]. It allows the development of server side applications fully in JavaScript by executing the code in Google V8 JavaScript engine, written in C++. There are several advantages in this approach:

- Node.js in combination with Express.js, a web framework for Node, has a natural affinity in building RESTful APIs;
- Use of only one programming language across all applications, both on server and client side, and to interact with the database (MongoDB);
- Node.js can perform all the I/O operations asynchronously, increasing the application throughput and scalability;
- Node.js can work with data streams over HTTP, which could help to implement a family of streamable WPS. This idea was not implemented in this thesis and is left as future work.

The disadvantage of Node.js is that JavaScript still does not have mature geospatial processing libraries, needing to rely on Turf.js [136], developed by Mapbox to do simple spatial operations in the browser; JSTS Topology Suite [137], a port of the well-known Java library JTS Topology Suit; and node-gdal [138], which provide Node.js bindings to the native GDAL library bindings in C++. All libraries are in the initial stage of development, not being robust or having a high variety of spatial functions.

The primary data format used in the services is JSON-LD. For spatial data is used ExtGeoJSON, an extension of GeoJSON presented in Chapter 6 extended to be consistent with JSON-LD. We store the data in a MongoDB database, a free and open-source document-oriented database, which does not uses table-based structures such as in a relational database, but uses JSON-like documents which allow dynamic schemas. MongoDB is a perfect fit when working with JSON files since it is its native structure, only having the overhead of converting to the native binary formatted used by the database, BSON. Also, MongoDB is a NoSQL database which uses JSON-like objects to specify queries. The disadvantage of MongoDB is that it has limited geospatial functionality compared to PostGIS, having a small number of spatial function, only natively support WGS84, and does not support R-trees for spatial indexing. Nevertheless, most of the spatial queries performed against the database in WFS and CSW are simple, such as searching within bounding boxes and performing spatial relations. The library Mongoose [139] provides the communication between MongoDB and Node.js.

For the implementation of WebSockets, we used Stream.IO [140], an open-source JavaScript library that provides both the client and the server side implementation needed for WebSockets communication. For all services was created a socket which broadcasts if the metadata has changed. As discussed in Chapter 2, Catalogue Services can exploit this connection to make sure the metadata is always up-to-date.

A common problem in using data from multiple origins of data, such as in a composition, is the limitation of cross-origin HTTP request. For security reasons, browsers do not allow the request of certain types of data within a script tag (via JavaScript that comes from another web server domain). One common solution is the use of JSON-P (JSON with padding) format [141], which is not desirable since is a different format, having a different Internet Media Type, and needs to be handled by the client in a different way than plain JSON. A more modern and flexible solution is the use of Cross-Origin Resource Sharing (CORS) [142], which is a standardized set of headers implemented in all modern browsers which allow the server and the client to communicate which requests are allowed for cross-domain requests. In this thesis, all the implemented services support CORS.

Whenever a service receives JSON data, for example in an update or inserts requests in WFS, as input data for a process in WPS, or as a query in JSFE in a CSW, it requires to the JSON file be validated. As discussed in Chapter 2 the JSON file is described by a JSON Schema version 4, which allows the validation of the expected keys and correspondent types. The schemas are maintained by the services and are accessible to the client. To validate the JSON file against the schema, we used the Tiny Validator for v4 JSON Schema (Tv4) [143].

In the next sections is discussed specific operations of each service type.

7.2.1 Web Feature Service

We define nine operations in a WFS: Get Capabilities, Describe Feature, Get Feature, Get Feature by ID, Insert, Update, Delete, Ad hoc queries, and Batch Processing.

In the Get Capabilities operations, the client can retrieve the metadata from the service. A client performs a HEAD request to the root URL of the service and follows the link relation *http://www.w3.org/ns/hydra/core#apiDocumentation*, specifying the Accepts header as *applica-*

tion/ld+json. The service returns the HTTP status code 200 (OK) and a JSON-LD document based on the Hydra Core Vocabulary discussed in Chapter 6. From this document, the client can access information from the service metadata, the available resource, and for each resource the specific metadata, JSON-LD context, and JSON Schema. In the case that the client wants to access the context and the JSON schema of a specific resource without the need to parse the metadata document it can use the Describe Feature operation, by performing a HEAD request to the specific URL of the resource and follow the link relation *http://www.w3.org/ns/json-ld#context* in the case of JSON-LD context, the link relation *http://json-schema.org/json-schema* in the case of JSON schema, and the link relation *http://www.opengis.net/rest-ows/resourceMetadata* to retrieve just the metadata correspondent to that resource.

In the Get Feature operation, the client retrieves a feature collection from a resource. A client performs a GET request to a specific URL of a resource with an acceptable value for the Accepts Header. The client also can specify query parameters in the URL, such as BBOX, attribute projection, start index, sort by, and count. In the case of success, it returns the HTTP status code 200 (OK). In a similar way, in the Get Feature by ID operation, the client retrieve a specific feature with known identifier (ID). A client performs a GET request to a specific URL of an ID of a resource with an acceptable value for the Accepts Header. In the case of success, the server returns the HTTP status code 200 (OK).

In a transactional WFS, the Insert operation inserts one or more features into a resource. A client performs a POST request to a specific URL of a resource with a payload containing the feature instance or feature collection in ExtGeoJSON, and the Content-Type header as *application/ext.geo+json*. The server performs a two-step validation of the file, by first verifying against a generic ExtGeoJSON schema and then verifying against the specific schema of the resource. In the case of success the features are inserted in the database, returning the HTTP status 201 (Created) and a Location Header with the URL of the resources created.

In the Update operation, the client can update one specific feature with known ID from a resource. A client performs a PATCH request to a specific URL of an ID of a resource with a payload containing the feature instance in ExtGeoJSON and the Content-Type header as *application/ext.geo+json*. The server performs the validation in the same way as described in the Insert operation and then updates the feature in MongoDB returning the HTTP status 200 (OK) in the case of success.

The last operation of Transactional WFS is the Delete, where the client can delete one specific feature with known ID from a resource. A client performs a DELETE request in a specific URL of an ID of a resource. The server verifies if the resource exists and deletes the feature in MongoDB returning the HTTP status 204 (No Content).

For WFS that supports Ad Hoc Queries a client can perform a POST request to the *queries* resource with a payload containing a query encoded in JSFE and specifying the Content-Type header as *application/ld+json*. The server performs the validation against the JSFE schema, and in the case of success returns the HTTP status 202 (Accepted) and a Location Header with the URL to the results of the query. The client then can perform a Get Feature operation to retrieves the results. Since the results are also a common WFS resource, it is indistinguishable from a regular layer, which allows the use of query parameters and Get Feature by ID operation.

The Batch Processing was not implemented in this thesis. However, for WFS that supports Batch Processing a client can perform a POST request to the *transactions* resource with a payload containing a Multipart MIME message specifying the transactions that are needed to be performed. The Content-Type header is *multipart/mixed*. The service returns a multipart message to the client as a response, specifying the result of each operation individually.

In case of failure different status codes are returned depending on the situation:

- 404 (Not Found): the client requests an invalid resource or ID.
- 406 (Not Acceptable): the client requests a resource representation that is not available. For example, if a client during content negotiation requests the resource to be provided in XML when the service is not able to do so.
- 415 (Unsupported media type): the client performs a POST or PATCH operation sending a resource in a media type that is not acceptable to the server. For example, if a client tries to insert a feature serialized in XML when the service is not able to do so.
- 422 (Unprocessable Entity): when the file sent to the server is malformed or failed the validation against a schema.

By the use of content negotiation the client can choose whether he wants to visualize the data in the browser or to access directly the data. The implementation is available at GitHub.[1] Figure 7.1 gives an example of a GetFeatureById request being performed in the browser, returning the visualization of the feature.

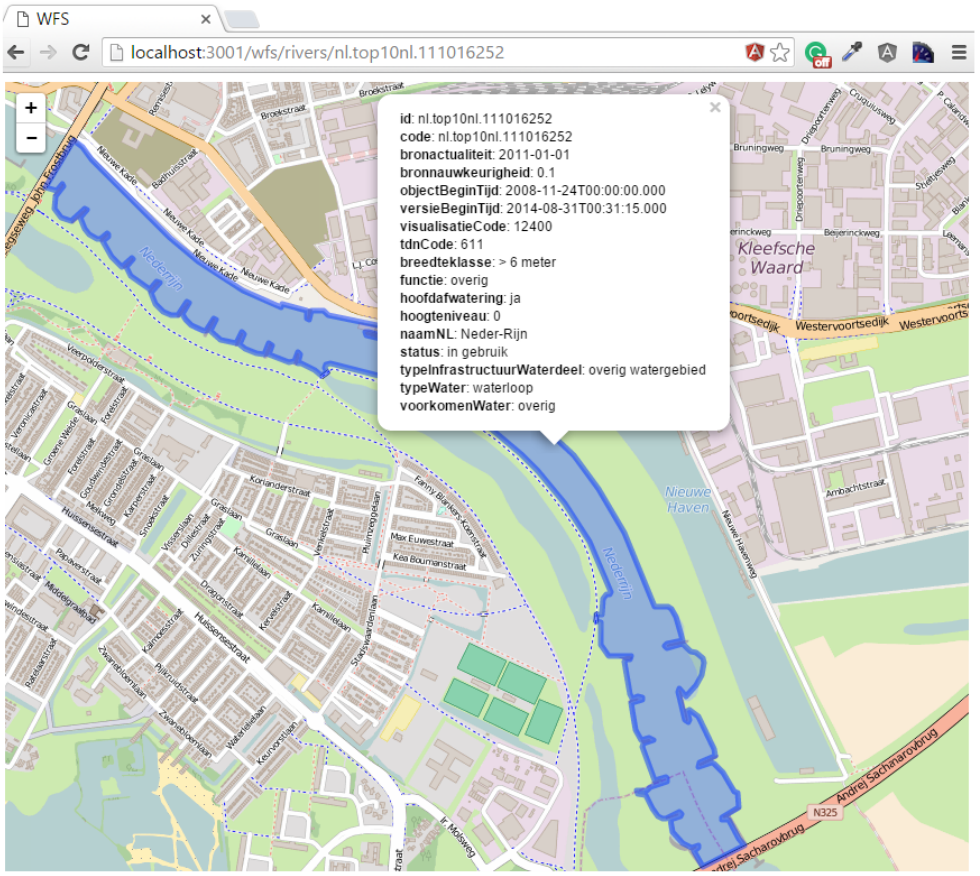


Figure 7.1: Content negotiation in the browser returning visual representation

7.2.2 Web Processing Service

We define seven operations in a WPS: Get Capabilities, Describe Process, Describe Result, Execute, Get Status, Get Result, and Dismiss Job.

[1] https://github.com/dinizime/wfsjs

The Execute operation allows the client to execute one process by performing a POST requests to the URL of the process with a payload containing a JSON-LD file following the standard input format for WPS and the Content-Type header *application/ld+json*. The server first validates the file against the input schema; then it verifies if any of the inputs contains data passed by reference. In this case, the service needs to request the inputs, which is done in parallel. If there are one or more inputs by reference, they are validated against the input schema. If all inputs are valid, the server creates a Job and returns the status code 202 (Accepted) to the client, with a Location Header for verifying the status of the job. Listing 7.1 presents an example of an WPS input, where both inputs (*filter* and *ftr*) receives data passed by reference, coming from one WFS and from a result from another WPS process.

Listing 7.1: WPS input example

```
{
  @context: {
    wps: "http://www.opengis.net/ont/wps/",
    inputs: "wps:input",
    ftr: {
      @id: "http://www.opengis.net/ont/spatial/object#SpatialObject",
      @type: "@id"
    },
    filter: {
      @id: "http://www.opengis.net/ont/spatial/object#SpatialObject",
      @type: "@id"
    }
  },
  @type: "inputs",
  filter : "http://localhost:3000/wps/buffer/jobs/job:c7645010-d0bb-11e5-bdc4-
      edb06f859154/outputs/buffered",
  ftr: "http://localhost:3001/wfs/buildings?bbox=5.902,51.949,5.947,51.984"
}
```

With this URL, the client can perform a Get Status operation with the Accept header *application/ld+json*, where it receives a JSON-LD file containing the status of the current job and optionally a time estimation of completion. The job status follows the WPS standard, with the possible values: *Succeeded*, *Failed*, *Accepted* (valid input, however, the server still did not start the job), and *Running*. When the status is succeeded the server also returns an additional property *resultsUrl*. This property contains the URL necessary for the Get Results operation, which the client can perform a GET requests with Accept header *application/ld+json*, and retrieve a Results document. Listings 7.2 and 7.3 presents an example of a Get Status response and of a Get Results operation, respectively.

Listing 7.2: WPS status example

```
{
  @context: {
    wps: "http://www.opengis.net/ont/wps/",
    job: "wps:job",
    status: {
      @id: "wps:status",
      @type: "@id"
    },
    jobID: "@id",
    expirationDate: "wps:expirationDate",
    estimatedCompletion: "wps:estimatedCompletion",
    nextPoll: "wps:nextPoll",
    percentComplete: "wps:percentComplete",
    resultsUrl: {
      @id: "wps:resultsUrl",
      @type: @id
    },
    Succeeded: "wps:status/succeeded",
```

```
    Failed: "wps:status/failed",
    Accepted: "wps:status/accepted",
    Running: "wps:status/running",
},
    @type: "job",
    jobID: "job:e08cfa40-d0bd-11e5-87ad-e17a6c7ed164",
    status: "Succeeded",
    resultsUrl: "http://localhost:3000/wps/intersects/jobs/job:e08cfa40-d0bd-11e5-87
        ad-e17a6c7ed164/outputs",
}
```

Listing 7.3: WPS result example

```
{
  @context: {
    wps: "http://www.opengis.net/ont/wps/",
    outputList: "wps:outputList",
    passed: {
      @id: "http://www.opengis.net/ont/spatial/object#SpatialObject",
      @type: "@id"
    },
    failed: {
      @id: "http://www.opengis.net/ont/spatial/object#SpatialObject",
      @type: "@id"
    }
  },
  @type: "outputList",
  passed: "http://localhost:3000/wps/intersects/jobs/job:e08cfa40-d0bd-11e5-87ad-
      e17a6c7ed164/outputs/passed",
  failed: "http://localhost:3000/wps/intersects/jobs/job:e08cfa40-d0bd-11e5-87ad-
      e17a6c7ed164/outputs/failed"
}
```

The results document contains the identifier of each output with the correspondent URL that allows accessing the results. Each result resource behaves in the same way as a WFS, which allows the use of query parameters and Get Feature by ID. At this moment, the server also dynamically generates the metadata of the output by using the metadata propagation mechanism discussed in Chapter 6. The Describe Result operation allows accessing the metadata of the result, which is not part of the WPS standard. It behaves in the same way as Describe Feature specified for WFS, the client performs a HEAD request and follow the Links headers that lead to the JSON-LD Context, to the JSON Schema or the specific metadata of the result. Figure 7.2 presents the sequence diagram for a WPS execution. Figure 7.3 shows the result of the WPS process by accessing the result URL in the browser.

For a WPS that supports Dismiss Job operation, a client can perform a DELETE request to the URL of a process. The server verifies if the process exists, stopping it in the case it is running or deleting its results in the case it completed, returning the status code 204 (No Content). The operations Get Capabilities and Describe Process behaves in the same way as specified for WFS.

In the implementation of WPS we made use of the libraries Async.js [144], which allows the management of asynchronous control flow such as parallel, series and waterfall executions, and node-uuid [145] which enables the generation of universally unique identifier (UUID) conformant with RFC 4122 [146], which was used to give unique identifiers to the jobs.

Also, as discussed in Chapter 2, the WPS service manage WebSockets connections to inform the user of the current status of the job, to remove the need of a client to perform polling (sending requests at regular intervals) or to rely on the job ETA. When a client creates a job, is created a WebSockets connection with the same identifier as the job, which the client can connect and receive real-time information about the changes in status. Whenever the job is completed, the connection is closed, and the client can perform the Get Result operation. The implementation is

available at GitHub.[2] It contains a series of spatial functions from Turf.js with the necessary documentation for verifying the composability. It also implements the Orchestration WPS described in the Section 7.4.1, and the Composition Verification WPS described in the Section 7.4.2.

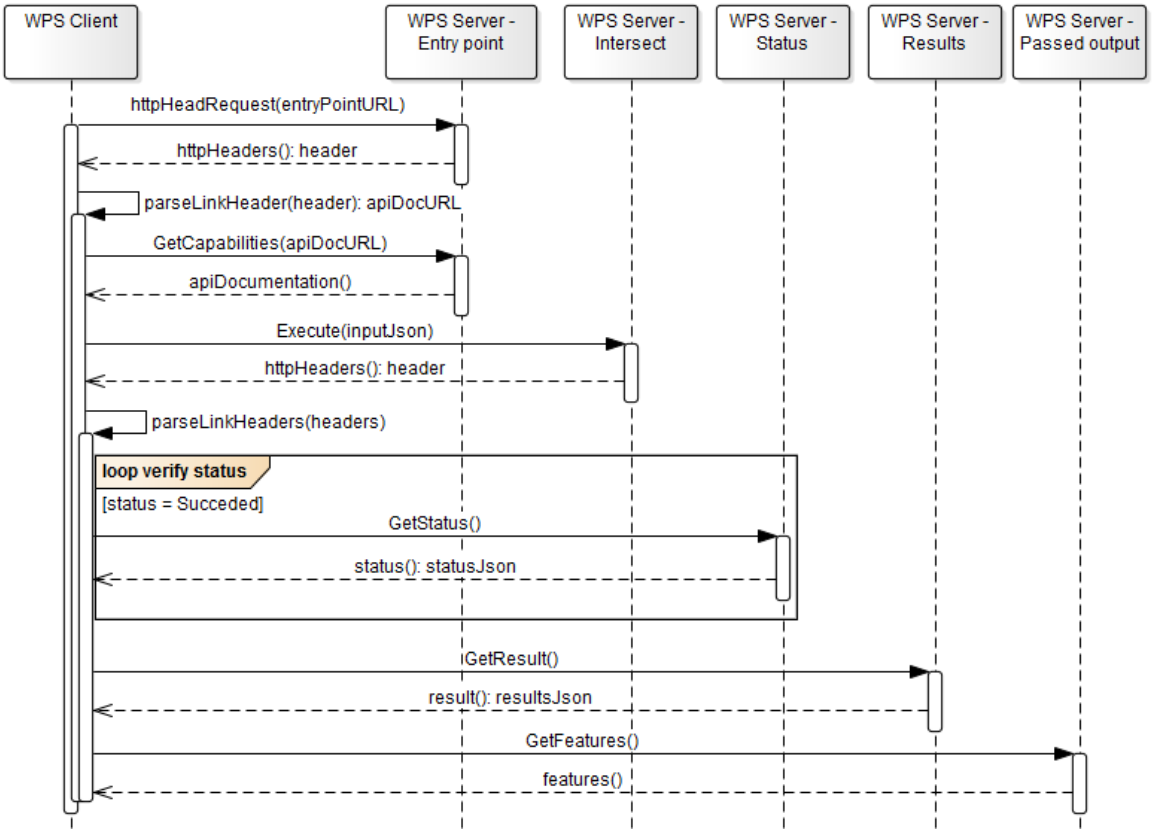


Figure 7.2: Sequence diagram for WPS execution

7.2.3 Catalogue Service for the Web

We define eight operations in a WCS: Get Capabilities, Describe Record, Get Records, Get Records by ID, Ad Hoc Queries, Get Domain, Delete, and Harvest.

The operation Harvest allows the catalogue to create or update records. It is started by a POST operation to the *harvest* resource with a payload containing a JSON-LD file with a list of services URL that the service will crawl. The Content-Type header is *application/ld+json*. The server first validates the input and then requests each of the services URL. If the input is valid, the service returns the status code 202 (Accepted) with a Location Header with the URL of the status of harvesting. The client can access this document to verify if each of the services was successfully harvested, what was the given identifier to each harvested record, and information specifying if the record was added or updated. The crawler expects the services to be described with the Hydra Core Vocabulary, as discussed in Chapter 6.

The operation Get Domain allows the client to retrieve run-time information about the range of values of an attribute of the records. This operation is useful when creating user interfaces and is needed to show the user the possible options to fill an attribute. The client can perform a GET

[2]https://github.com/dinizime/wpsjs

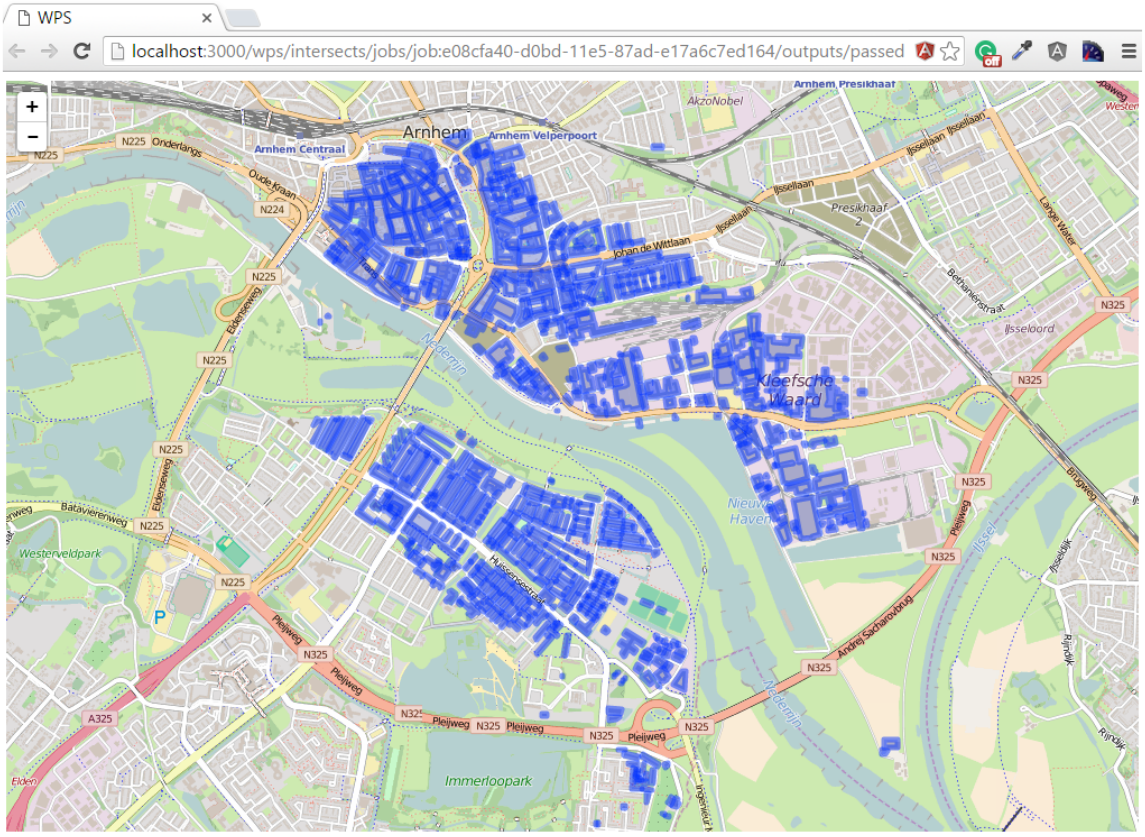


Figure 7.3: WPS result in the browser

request on the resource *domain* specifying by hierarchy the name of the attribute which the client wants to retrieve the range of values. If the attribute exists, the server returns the status code 200 (OK), with a JSON-LD list of the possible values.

The operations Get Capabilities, Describe Record, Get Records, Get Records by ID, Ad Hoc Queries, and Delete behaves in the same way as specified for WFS. Also, as discussed in Chapter 2, the CSW service is capable of connecting to multiple WebSockets of services, receiving information in case of a metadata update. This allows the CSW to manage efficiently the Harvest operation and to guarantee that always have the most up-to-date metadata.

As discussed in Chapter 2, the semantics of the records was also extended, which allows the storage of information other than service metadata. Section 7.4.3 specifies a profile of CSW that allows the storage of workflow information. The implementation is available at GitHub.[3] It also implements the Workflow CSW described in the Section 7.4.3.

7.3 GENERIC CLIENT

As a proof of concept, we implement a Generic Client (GC) capable of processing the Hydra Core Vocabulary, which allows the interface to be built according to the functionality of the service. It allows seamless interaction with all available operations of WFS, WPS, and CSW. This implementation supports the idea that the proposed services can be specified without standardization

[3] https://github.com/dinizime/cswjs

of URLs and with the heterogeneous availability of operations per resource. The goal is to assist human clients in the use of OGC services.

It was implemented using Angular.js [147] and Leaflet [148]. Also, we used Leaflet.draw [149] for the acquisition of geometries. The GC is capable of verifying which layers accepts transactional operations, enabling the acquisition of geometries with the correct types, and generation of forms to fill attributes, with built-in type verification. The GC is also capable of managing the execution of processes in a WPS, adding the results of resources. The implementation is available at GitHub.[4]

Figure 7.4 presents the Generic Client user interface accessing a WFS service.

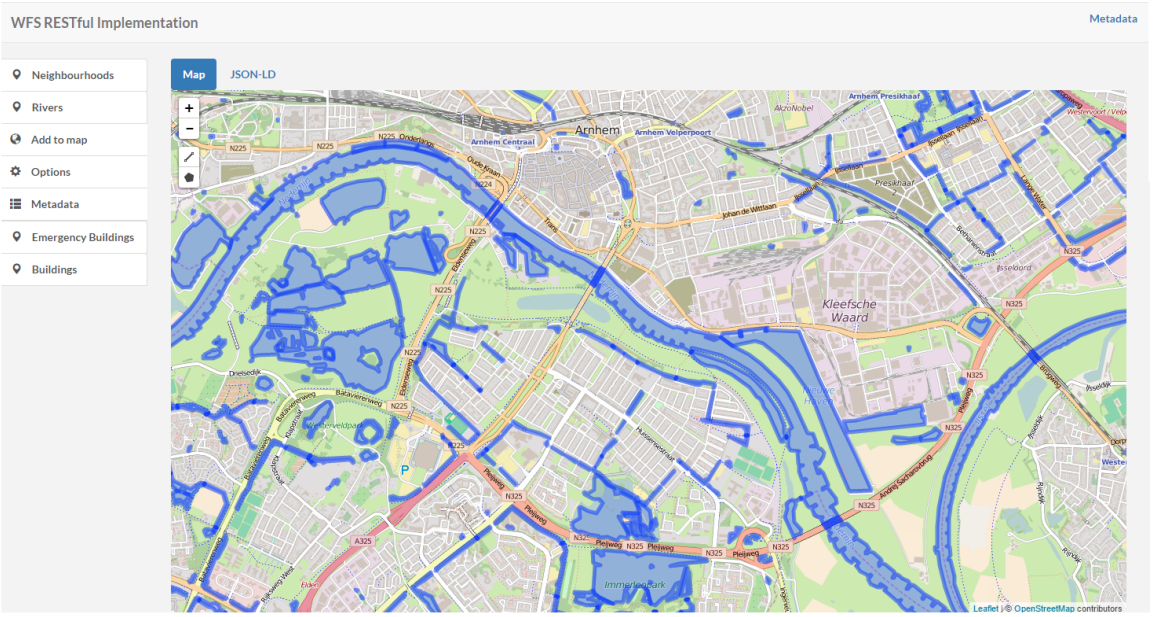


Figure 7.4: Generic Client user interface

7.4 SPECIALIZED SERVICES

In this section, we discuss three particular types of services that deserve a more in-depth discussion. The Orchestration WPS that allows the asynchronous orchestration of compositions; the Composition Verification WPS, that permits the verification of a composition according to the theory of Chapters 3, 4, 5; and the Workflow CSW, that is capable of storing and distributing workflows.

7.4.1 Orchestration WPS

In Chapter 3, we define web service composition as the process of selecting and assembling a meaningful combination of services to solve a specified problem that cannot be solved by individual services. The composition process is divided into synthesis, the selection of services and construction of workflow, and execution, coordination of the passing of data among the component services. Execution can be performed by orchestration or choreography. In *orchestration* exists a central service which coordinates the execution and passing of data between the services of the composition. In *choreography* there is no central services, each service is responsible for coordinating the

[4]https://github.com/dinizime/ogc-client

execution of the next. In this thesis we focus on an orchestration execution, since choreography requires a significant change in the WPS specification, requiring service functionality to be rather complex, going against the vision of this thesis.

In this section we propose a simple orchestration engine, which can be encapsulated in a WPS, and focus on the execution of the asynchronous mode of WPS, creating and monitoring jobs according to the specification. An Orchestration Engine (OE) receives as an input a composition description, and coordinate the services execution and the communication of data, returning the result to the client when complete. It should be able to do exception handling, and ideally, in the case of error, propose another workflow.

Walia et al. [150] uses the WPS interface to orchestrate multiple WPS and WFS services. This approach is rigid, as it does not allow the selection of services to be composable, and requires knowledge of the implementation of the different services to verify composition validity. Meng et al. [151] describe the application of three different approaches to service composition, making use of the WPS interface, cascading service chains from via WPS GET operations, and the Business Process Execution Language (BPEL). The downside of this approach is that the composition is specified in WSDL, requiring human interaction for every service composition. Also, all three approaches are rigid, and must be specified by a human upfront. Jesus et al. [152] does not use BPEL but instead describes the composition in WSDL and the Taverna Workbench is used to perform the orchestration of the services. This approach is more flexible than the former two but is required the use of the SOAP interface.

In Akram et al. [153] evaluates BPEL for the orchestration of scientific workflows. The work established a series of requirements in which we can evaluate our proposed orchestration engine:

- Modular design: it can reutilize workflows and break workflows in pieces with smaller complexity. The proposed OE can use the parametric workflow as a component in the composition.

- Exception handling: it can identify and return errors at run-time. Based on the proposed theory of composability we identify a group of errors that can exist during the composition execution. The OE is capable of raise exception for this group of errors.

- Compensation mechanism: it can undo steps that already completed successfully. This is not supported by the proposed orchestration engine, and in this thesis, we did not focus on transactions. However, all partial results of the orchestration are available, which allows the user to rebuild the process from a desired part of the composition.

- Adaptative workflows: it can redesign and optimization at run-time. This concept is related to service substitutability, presented in Chapter 5. The current implementation of the OE does not support this case.

- Flexible workflows: the client can modify workflows.By the use of the Orchestration Client presented in Section 7.5, the client can modify the workflows and store them in a Workflow CSW.

- Management of workflows: the client can monitor the execution of workflows, also allowing to stop and resume the execution, and to verify partial results. By the use of the Orchestration Client presented in Section 7.5, the client can monitor the execution of the workflow and access the results. However, the current implementation does not allow one to pause or stop the execution.

The orchestration engine allows the execution of components in parallel and focuses on the asynchronous execution of WPS. It is implemented as a WPS with as input a JSON-W, and it can build a request for job creation and monitor the job status. The orchestration engine first builds a dependency tree based on the topological sorting of the graph structure of JSON-W, which allows the identification of a proper service execution order. Also, it uses the library async.js to identify and run processes in parallel when possible. The OE performs two activities, build requests and track status. Based on the derived order of execution the OE builds the requests for WPS services. It does not need to communicate with WFS since the data will be indicated by reference to the WPS input. After the input is built the OE starts an Execution as described in Figure 7.2. The main responsibility at this stage is to track the status of the execution, and when it is complete request the location of the results. Figure 7.5 shows an example sequence diagram for the orchestration of the composition illustrated in Figure 6.1.

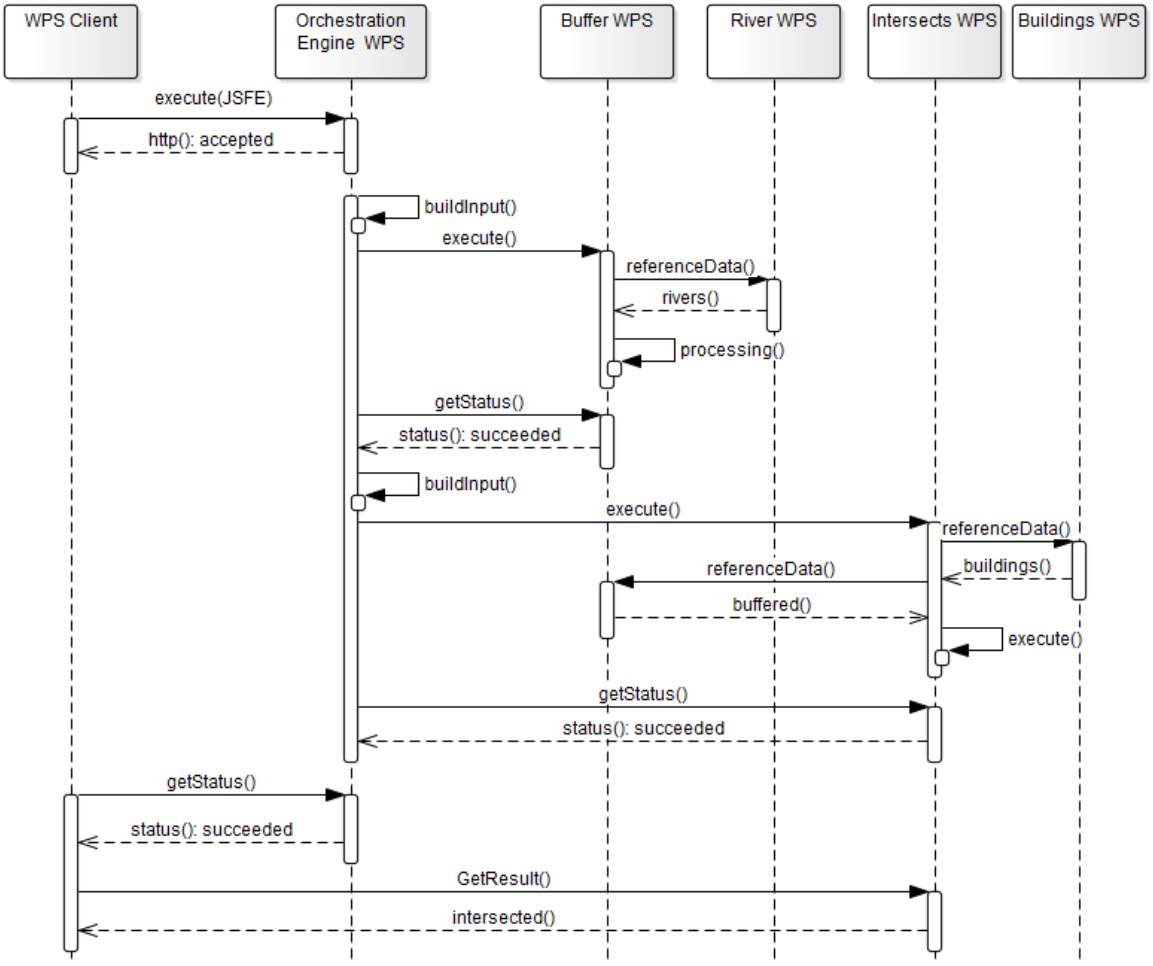


Figure 7.5: Example of orchestration execution

We added the attribute *orchestrationStatus*, a list of objects, to the status document so that the client can keep track of the status of the component services. The orchestration engine returns a document with the results of the process, also including partial results. Listing 7.4 exemplifies an status document for a orchestration execution (with the context omitted). Listing 7.5 exemplifies an orchestration result document (with the context omitted).

Listing 7.4: Orchestration status example

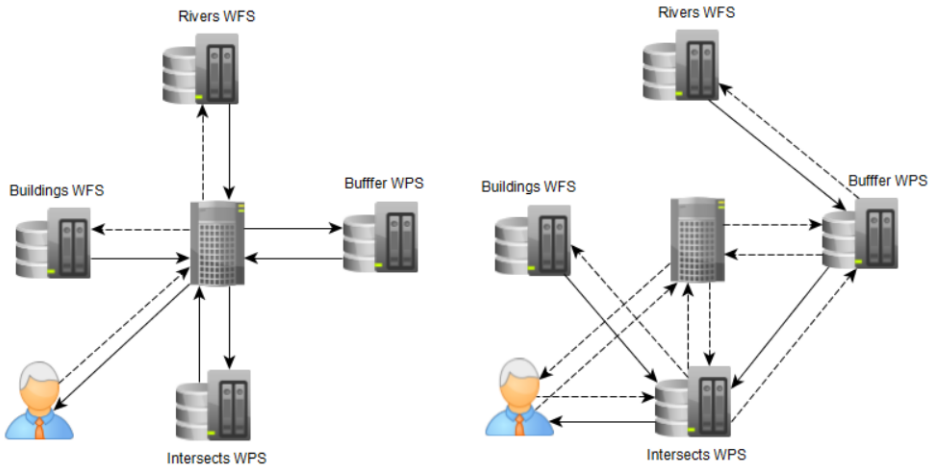
```
{
  @type: "job",
  jobID: "job:9331a500-d119-11e5-9dc0-43d8c9620c6e",
  status: "Running",
  orchestrationStatus: [{
      status: "Succeeded",
      name: "Literal",
      @id: "task:0"
    },{
      status: "Succeeded",
      name: "Rivers",
      @id: "task:1"
    },{
      status: "Succeeded",
      name: "Buffer",
      @id: "task:2"
    },{
      status: "Succeeded",
      name: "Buildings",
      @id: "task:3"
    },{
      status: "Running",
      name: "Intersects",
      @id: "task:4"
    }
  ]
}
```

Listing 7.5: Orchestration result example

```
{
  task:2: {
    buildings: "http://localhost:3001/wfs/buildings?bbox=5.902,51.949,5.947,51.984"
  },
  task:3: {
    rivers: "http://localhost:3001/wfs/rivers/nl.top10nl.111016252"
  },
  task:4: {
    literal: 1000
  },
  task:0: {
    buffered: "http://localhost:3000/wps/buffer/jobs/job:93430a20-d119-11e5-9dc0-43
        d8c9620c6e/outputs/buffered"
  },
  task:1: {
    passed: "http://localhost:3000/wps/intersects/jobs/job:94e8cae0-d119-11e5-9dc0-43
        d8c9620c6e/outputs/passed",
    failed: "http://localhost:3000/wps/intersects/jobs/job:94e8cae0-d119-11e5-9dc0-43
        d8c9620c6e/outputs/failed"
  }
}
```

The advantage of using the asynchronous execution of WPS is that by using the mechanism of data transfer by reference, it reduces the bandwidth usage by not requiring the orchestration service retrieve and send the data between services. This is only possible due to the modification of the WPS result to be indistinguishable from a WFS, harmonizing the passing of data between OGC services. Figure 7.6 compares the pass of data by value and by reference. The central server represents the orchestration engine, the solid lines represent data passing, and the dashed lines represents requests without data. By using data passing by value, the orchestration engine must request all the data and re-transmit to the appropriate service. Using data passing by reference the orchestration engine does not need to request for data, it only needs to construct the appropriate request, and the WPS server can request the needed data. It is expected that on average this

procedure reduces in 50% the data transfer.



(a) Data passing by value (b) Data passing by reference

Figure 7.6: Comparison of data passing by value and reference

Three cases require special treatment in the composition: subgraph, conditional, and loop. For handling subgraphs, the orchestration engine has two options. The first is nested execution, where whenever the orchestration engine needs to execute a subgraph it creates another orchestration instance that executes the subgraph, and when it is complete it passes the result to the main orchestration instance. The second is pre-flattening, where the orchestration engine identifies the existence of subgraphs, requests the necessary components, and rebuilds the workflow in a way that there are no subgraphs. The latter is simpler, requiring minor modifications to the orchestration engine. For conditional evaluation exists two possibilities: the data is transferred to the orchestration engine which evaluates the condition, or the orchestration engine can perform a query with an appropriate statement of the evaluated service, verifying if the condition is true or false. The latter is preferred, as there is no need for data transfer. Also, the orchestration engine verifies if the conditional node can be resolved at compile-time, removing unnecessary conditionals.

In the case of loop each of the introduced loop nodes needs to be discussed separately. In the case of Iterate Inputs and Iterate Multi-Value, the Orchestration engine only needs to manage the correct calling of services. To handle Iterate Sets, it is necessary a mechanism extract features instances individually from the set, or the orchestration engine need to retrieve the entire set from the service and send the instances to the iteration accordingly. In Chapter 2 was discussed how the output of WPS was modified to be indistinguishable from a WFS, meaning that the operation GetFeatureById can be applied to extract the inputs individually from the set. As the last step for Iterate Multi-Value and Iterate Sets, it is necessary the aggregation of the results. If the next step in the workflow accepts multi inputs, it is just required for the orchestration engine to aggregate the URL results in an array. If it does not accept multiple inputs, the data needs to be transferred to the orchestration engine, which aggregates the data and becomes a temporary access point.

7.4.2 Composition Verification WPS

Verification WPS is a common WPS with a process that can verify if a workflow representation is composable. It receives as an input a JSON-W describing a composition and performs a series of operations following the theory described in Chapters 3, 4, 5. It returns a JSON-LD document

specifying the errors found, if any. Figure 7.7 shows the general flowchart of the operations that are applied during the verification of the composition.

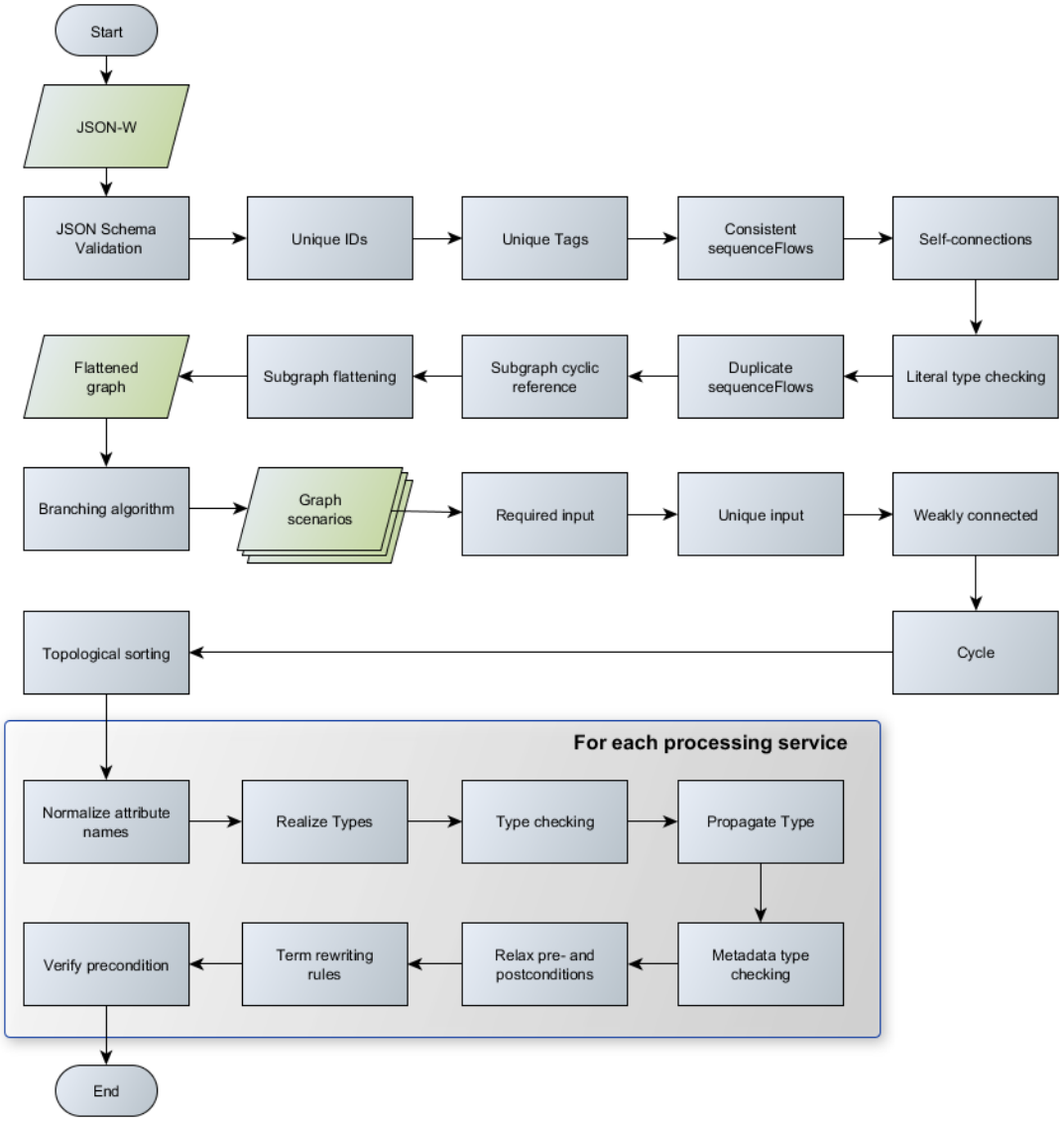


Figure 7.7: General flowchart for composition verification

The verification of a level of composability can only be executed if the workflow has enough metadata for its verification. Each of the applied algorithms has a correspondent error code, that can be used by the user to identify the problems, or used by an application to take further actions. Listing 7.6 exemplifies an result of the Composition Verification WPS (with context omited).

Listing 7.6: Composability result example

```
{
  tasks: {
    "task:6": [
      {
        @type: "taskCycles",
        message : "Cycles are not allowed",
        cycles: [
          [
```

```
            "task:6",
            "task:9",
            "task:8"
          ]
        ]
      }
    ],
    "task:8": [
      {
        @type: "invalidType",
        message : "Connection did not type check",
        inputType: {
          "set": {
            "record": {
              "geom": "Point"
            }
          }
        },
        outputType: {
          "set": {
            "record": {
              "geom": "Polygon"
            }
          }
        }
      }
    ]
  },
  sequenceFlows: {}
}
```

7.4.3 Workflow CSW

The Workflow CSW is a profile of a CSW that can store two types of records: workflows and parametric workflows. The workflows are serialized in JSON-W, which makes convenient its storage in a document database, such as MongoDB. The workflow metadata can be queried with the operations Get Records, which allows workflow discovery.

The workflow record stores the workflow JSON representation and its basic metadata. This enables the execution of the workflow by sending it directly to an Orchestration Engine, which can be done by reference by using the URL of the GetRecordsById. The workflow can also be requested, modified, and updated in the CSW. This can be done manually or by the use of the Orchestration Client discussed in the next section.

The parametric workflow record stores the workflow as an opaque composition, which can be used as component service in a composition. It follows the same structure as in the JSON-W, being able to embed the workflow information or just reference it in the URL attribute.

7.5 ORCHESTRATION CLIENT

As a proof of concept, we implement the Orchestration Client (OC) to assist a human client to build, verify and execute compositions. The interface gives the user a visual way of creating the JSON-W representation. The OC can connect directly to WPS and WFS or connect to a CSW, which allows interacting with all the services stored. It also permits the connection to a Workflow CSW, where it can discover and load workflow records, for execution or modification, or parametric workflow records, which allows one to use workflows as components in a composition. This connection with the Workflow CSW is transactional, which allows the client to create, update, and delete both workflows and parametric workflow records. This creates an easy was of sharing

a composition in an interoperable manner. This interface is also capable of dealing with special nodes such as a conditional and controlled loop. The implementation is available at GitHub.[5]

Figure 7.8 presents the interface of the OC.

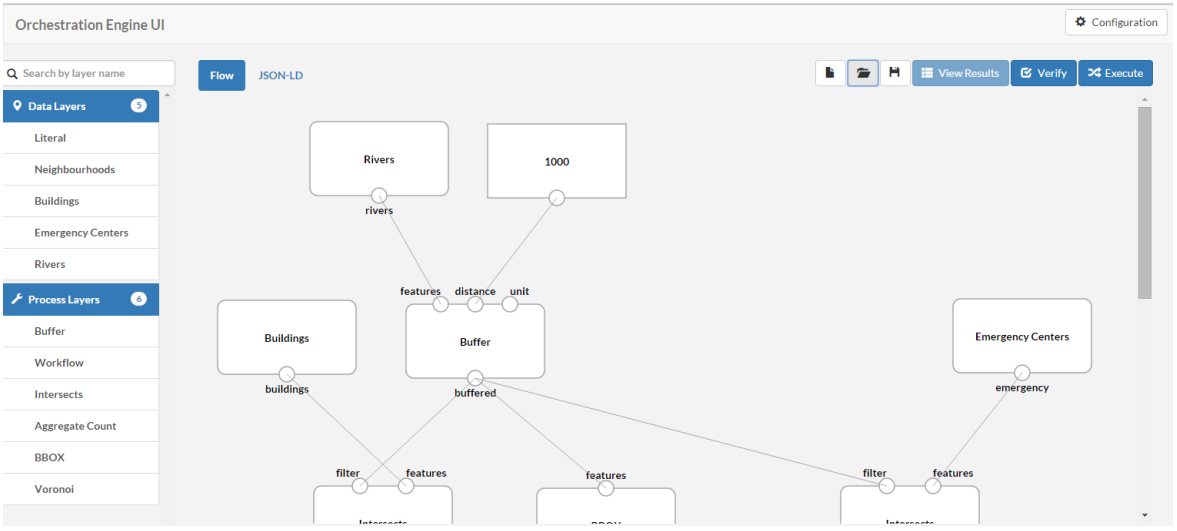


Figure 7.8: Orchestration Client user interface

When the client completes the construction of a composition, it can send it directly to Composition Verification WPS, where the interface allows the client to follow the progress of the verification and display the results both in the JSON format and by highlighting the errors in the interface. The OC client also can perform some of the verifications on-the-fly, not needing to send to a verification WPS. If the composition has no errors the user can send it directly to an Orchestration WPS, where the client is capable of monitoring its execution, and to access the results. Figure 7.9 presents the error highlight in the OC.

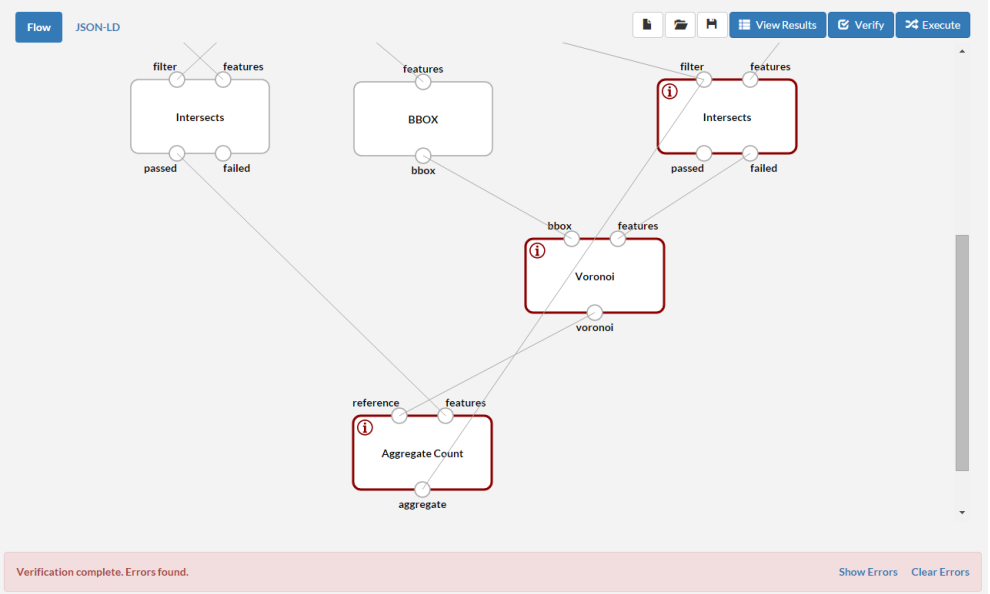


Figure 7.9: Orchestration Client error highlight

[5]https://github.com/dinizime/orchestration-client

7.6 SUMMARY

In this chapter we describe the implementation of the proposed RESTful services, giving a more in-depth discussion about three specialized services, namely, Orchestration WPS, Composition Verification WPS, and Workflow CSW. We show that is viable the implementation of OGC web services purely in JavaScript server side (Node.js). However, the state of current implementations of spatial libraries in JavaScript is still in its infancy. Also, document database fits well for the use of JSON documents, not creating an overhead for converting to and from JSON objects. However, still the spatial capabilities of documents libraries need to be developed further to be able to match relational databases such as PostgreSQL/PostGIS. In the Orchestration WPS, we provide an implementation that can execute WPS asynchronously, and as well handle conditional, subgraph, and loop. Also, we show how the mechanism of passing data by reference can optimize the bandwidth usage of a service composition execution. In the Composition Verification WPS, we encapsulate in a WPS the concepts presented in Chapters 3, 4, and 5, which provides a compile-time verification of the composition. The Workflow CSW allows the storage and distribution of workflows both for direct consumption (running workflows without modification) and for using the workflow as an opaque composition that can be a component in another workflow.

Lastly, as a proof of concept, we implemented two user interfaces, the Generic Client, and the Orchestration Client. In the Generic Client implementation, we show that the Hydra Core Vocabulary can describe web service capabilities with a high granularity, which allows Hydra-aware applications to discover and use web services without being specifically programmed for each service, lowering human interaction. The Orchestration Client assists the user to construct JSON-W description by providing a graphical interface where the user can build the composition visually. The interface also can connect to web services to retrieve their metadata, to a Composition Verification WPS, which allows the verification of the workflow at compile-time, and to connect to an Orchestration WPS, which enables the execution of the workflow.

Chapter 8
Conclusions and Recommendations

8.1 CONCLUSIONS

OGC standards have made significant progress towards syntactic interoperability of services and feature-level data access. However, they do not solve semantic heterogeneity problems. At the same time, Semantic Web Technology can provide semantic interoperability but has had a slow uptake, since it is complex and does not follow current trends for data formats and access, thereby heading in an opposite direction of what developers and users expect. Even for services that provide semantically enabled data, service discovery and manual composition in a distributed environment are still time-consuming and error-prone. Typically, one must select multiple data provision services, and apply the processes with the need to understand their functionality and input/output restrictions. As a consequence of these difficulties, there is a lack of tools and methods to facilitate the construction, verification and execution of composition of geospatial web services.

In this thesis, several advancements are made in the fields of composition verification, sharing, and execution. For composition verification, four theoretically founded levels of composability are discussed and implemented. The focus of the theory of composability is compile-time verification, which prevents to run an invalid composition that could lead to unnecessary long running time as well high usage of the server processing capabilities. For composition sharing, we develop an interoperable JSON-LD workflow representation. It is capable of describing the interaction between web services, as well as that it covers the use cases of subgraph, conditional and loop, and capable of being stored and shared using a CSW. For composition execution, we develop an orchestration engine, implemented as a WPS, and able to executing WPS processes in asynchronous mode, which reduces bandwidth usage by not requiring the orchestration service to handle the data between services. We implemented all proposed services, including three specialized services: the composition verification WPS, the orchestration WPS, and the Workflow CSW. Also, we implement a generic OGC client capable of dynamically build user interfaces based on the available functionality of the service, and, to assist the use of JSON-W, we implemented an orchestration client, a graphical interface that allows the user to build the composition visually.

Although this research project modified the OGC services significantly, all the modifications are aligned with current OGC vision. Three engineering reports released by OGC were the base for the proposed modifications. *Testbed 11 REST Interface Engineering Report* [35] gives a first discussion about the ubiquitous use of RESTful bindings in OGC services. *Testbed 11 Implementing JSON/GeoJSON in an OGC Standard Engineering Report* [24] addresses the use of JSON, GeoJSON, and JSON-LD in multiple OGC services. *Testbed 11 Implementing Linked Data and Semantically Enabling OGC Services Engineering Report* [97] addresses the possible solutions to add semantic information to OGC services, being one of the conclusions that the most straightforward solution for semantically enabling OGC services is to consider RESTful services. All these documents were released in late 2015, showing that all issues discussed in this thesis are current and relevant problems in the geospatial community.

Furthermore, from the six areas of interest of the OGC Workflow Domain Working Group [154],

we address four in this thesis:

- Cost reduction of workflow implementation: in this thesis we modified the output of an asynchronous WPS execution to be indistinguishable from a WFS resource, harmonizing the passing of data between OGC services. Also, the service encodings were restricted to the proposed RESTful services.
- Workflow languages: in this thesis we propose a lightweight workflow language serialized in JSON-LD.
- Workflow interfaces: we implemented an orchestration client, which assists the user in building compositions and allows to monitor the progress of the orchestration.
- Architecture discussion: in this thesis, we implemented an orchestration engine that uses a mechanism for passing data by reference, which allows bandwidth usage optimization. Also, this orchestration engine uses the asynchronous execution mode of WPS. Moreover, we defined a mechanism of metadata propagation that allows the generation of workflow metadata.

In the paragraphs below, we answer the research questions.

1. How can current standards be modified to align with contemporary web technology?

   Chapter 2 discusses how REST and JSON are the most used technology in Web APIs, and we presented RESTful Binding for the WFS, WPS and CSW services discussing trade-off of applying these changes to the architecture. The use of REST provides a simpler alternative to the standard KVP and SOAP bindings in OGC services, and resource-oriented architecture provide scalability to the server, and also renders the services lightweight compared to RPC/SOAP/XML implementations, especially when combined with JSON as the main data format. The proposed services are flexible since they allow an individual definition of methods and parameters for each resource, and as the application needs. We also provided two scenarios in which the WebSockets protocol is applied in the OGC services, complementing the cases where REST architecture leads to an inefficient implementation. WebSockets can be used to enhance the capabilities of a CSW to keep metadata up-to-date, and to lower the overhead of communication when tracking the progress status of a job in an asynchronous execution of WPS. Lastly, we discussed the use of JSON as a data format for sharing metadata and spatial data in OGC services. We addressed the limitations of GeoJSON for sharing spatial data compared to GML and proposed solutions with JSON Schema and JSON-LD.

2. Which forms of composition can be recognized, and for each of these, to what extent can we decide on their validity?

   Chapter 3 discusses the division of the theory of composability based on the expressiveness of checking into five levels, namely, Structural, Static Syntactic, Dynamic Syntactic, Semantic, and Qualitative. Each of the levels gives a definition to service and service composition and specifies which types of error can be identified, and how to detect those errors.

   Structural Composability abstracts the composition as a directed graph and defines under which conditions this graph is valid. In this graph, services are represented as nodes and edges represents the passing of data between output and input. We discuss this level in Chapter 3. Static Syntactic Composability states that the output of a component service can only become the input of the next service if the respective data types are identical or the output

data can be automatically coerced to the data type of the input parameter. This level of composability aims to prevent type errors and is discussed in Chapter 4. Dynamic Syntactic Composability complements Static Syntactic Composability by evaluating restrictions related to data values that cannot be expressed in the type system. These include interaction between inputs, coordinate system, geometry dimension, time zone, and pixel size. We discussed this level in Chapter 5. Semantic Composability addresses the meaning of data and processes, verifying whether the combination of processes can derive a meaningful result, or whether it produces the intended result of the user. We discuss this level in Chapter 5. Qualitative Composability evaluates of the composition against user requirements related to non-functional characteristics such as response time, service availability, cost, security, legal rights, and quality of operations. This level of composability is not discussed in this thesis since it fundamentally is not related to protection against errors, but with the best selection of services to fulfill the user specification.

3. How to verify the validity of the composition of services?

   Throughout Chapters 3, 4, and 5 we present a series of methods to verify the composition of services. In Structural Composability, the composition is seen as a graph and describes under which conditions the graph is well-formed, meaning its execution does not produce structural errors. Examples of structural errors discussed are dependency cycle, self-connections, repeated tags within one service, non-weakly connected graph, a cyclic reference in subgraphs, unique and required condition violation for inputs.

   In Static Syntactic Composability, we evaluate service compositions using the Cardelli type system $F_{1<:}$, augmented with an additional type, the service type. This type is a special case of the function type, which allows one to express a service as a function with zero or more named inputs and one or more named outputs. We also defined abstract data types, to express concepts about geometry, coverage, and time, and type propagation operators, which allows one to express the output types of services as parameterized by the input types. We also discussed how to type-check compositions with subgraph, conditional, and loop nodes.

   In Dynamic Syntactic Composability, we use an extension of Hoare Logic that relates the inputs and outputs of a service with its pre- and postcondition. We define the most common variables used in geospatial services and five operations that allow the manipulation of pre- and postcondition on the basis of the interaction between inputs and outputs of the web services.

   In Semantic Composability, we divide the verification into four categories: input/output (IO), attributes, service, and workflow. In the IO category, an ontology is associated with the inputs and outputs of the service to prevent unexpected behavior of processes at the thematic level. This is done by restricting the input semantics to have the relationship *equivalentClass* or *subClassOf*. In the attribute category, an ontology is associated with attributes, and we present a mechanism that normalizes the attribute names across the composition by verifying the *equivalentProperty* relation. In the service category, processing services are associated with an ontology based on their functionality. We based the ontology on a geo-operators thesaurus. This category allows substitutability of services. A method for the workflow category is not provided in this thesis.

4. How can OGC services be extended to be semantically interoperable?

   Chapter 5 uses JSON-LD to relate metadata and ontologies, also allowing a lossless roundtrip to RDF model. The basic ontology for OGC services metadata is also provided. For geospatial data, GeoJSON cannot be used as a JSON-LD directly due to the impossibility of handling the conversion of a list of lists to the RDF model (the geometry of GeoJSON is encoded as a list of lists). In this way, we present ExtGeoJSON, an extension of GeoJSON format that allows multiple geometry representations, making it compatible with JSON-LD. Also, it allows identifying the semantics of the feature layer, of attributes, and of the geometry.

5. How can geospatial web services functionality be described in a machine-readable and machine-interpretable way?

   Chapter 6 enhances the metadata of the OGC services with the Hydra Core Vocabulary. It allows the description of the functionality of each resource of a RESTful service. This vocabulary is serialized in JSON-LD, which allows the metadata to be interoperable. By using Hydra, the OGC Web Services can better integrate with other RESTful APIs. We also improved the descriptions to contain information that allows verification of the composition. Those include service types, preconditions, postconditions, the semantics of the service, and semantics of the inputs and outputs. Chapter 7 describes the implementation of a generic client for OGC services capable of processing Hydra-enabled metadata and dynamically generating the user interface based on the functionality of the service. This client works seamlessly for WFS, WPS and CSW services.

6. How can the metadata of individual services be combined and propagated to generate descriptions for a composite service?

   Chapter 6 presents a metadata propagation method that allows generating the metadata of WPS results dynamically on the basis of input metadata. The propagation occurs in three different steps: generation of JSON Schema, generation of JSON-LD context and generation of general metadata. The former two allows the result of a WPS process to have a consistent metadata compared to a WFS. The latter can be used for provenance information and other metadata related to composition verification, such as postcondition, service type, and semantics. The same method can be used to generate the metadata of component services in a composition, which allows the generation of workflow metadata. This method is especially useful to describe parametric workflows, which allows them to be used as a component in a composition in a way that is indistinguishable from a WPS. The implemented metadata propagation mechanism is constrained to the generation of metadata necessary for the verification of composability.

7. How can geospatial service composition be described and shared in an interoperable way?

   Chapter 6 presents JSON-W, a JSON-LD object which describes service compositions, including conditional, subgraph, and controlled loop. We based this notation on the graph representation of a composition formalized in Chapter 3 and serialized in RDF in Chapter 6. By the use of JSON-LD, we maintain the ease to use and transfer to the Web of the JSON format while allowing lossless roundtrip to the RDF model. In the RDF model, the classes and properties are formalized by an ontology introduced in Chapter 6, which guarantees that the workflow description is interoperable. We describe two types of compositions: workflows and parametric workflows. The latter allow the workflow to be represented and used as a WPS in an orchestration engine, which in turn allows the use of the workflow as a component in a composition. In Chapter 7, we use this notation as the standard input for

the implemented orchestration WPS and verification WPS. In the same chapter, we present the Workflow CSW, a profile of CSW that is capable of storing and sharing workflows and parametric workflows. To assist the use of JSON-W, we implemented a graphical interface that allows the user to build the composition visually. This application also allows sending the workflow notation to a verification WPS, which assesses the composability of the workflow, and to an orchestration WPS that executes it.

8. How can bandwidth usage in the execution of a composition be minimized?

   Chapter 2 modifies the asynchronous execution of WPS processes in such a way that the result is indistinguishable from a WFS resource. Consequently, data access in OGC standards is harmonized, which allows consistent passing of data by reference instead of by value. Chapter 7 presents the Orchestration WPS, an orchestration engine encapsulated in a WPS that receives a workflow representation described in JSON-W and executes the composition. We implement it in such a way that it executes the component services in the asynchronous execution mode of WPS, individually monitoring their status, and controlling the overall composition in parallel and asynchronously. The passing of data by reference is exploited such that there is no need for data transfer to the orchestration engine. The orchestration engine is just responsible for building valid requests indicating the location of the inputs for a service, and the service itself is responsible for fetching the required resources. This mechanism significantly reduces data transfer during the orchestration.

8.2 LIMITATIONS

In this section, we mention some limitations of this research.

- Although in this thesis we implemented all the proposed methods, and we annotated Turf.js library functions with the metadata necessary for the composition verification, it is still necessary to evaluate its applicability in a real world workflow. This evaluation can help to understand for which cases the theory needs to be extended.

- The proposed JSON Workflow representation is tightly coupled with the proposed RESTful OGC services, and it is not compatible with KVP/SOAP encodings. Also, due to the JSON oriented approach, it is not compatible with W3C WS in general.

- Orchestration engine relies on the asynchronous mode of WPS, making it impossible to execute in synchronous mode. The Orchestration WPS needs to be extended to handle this case.

- Composability theory at all levels is not compatible with transactions, prompting the need to verify how transactions (atomic or not) can affect the verification of a composition.

- Compile-time verification cannot identify all errors. As we discuss both for Static Syntactic Composability and Dynamic Syntactic Composability, one cannot identify all errors at compile-time, requiring additional run-time verification.

- Metadata propagation does not allow to track provenance. In this thesis, the metadata propagation focuses on the elements necessary for the composition verification. The metadata propagation method needs to be extended to track information about which steps the workflow took to generate the result.

- JSON-W does not cover data formats and service bindings. The JSON-W notation assumes that the services are RESTful and JSON-LD based. One needs to extend the notation with sufficient information in a way that the orchestration engine can mediate between different data formats and service bindings.

8.3 SUGGESTIONS FOR OGC STANDARDS

In this section, we expose a list of recommendations for further testing and possible adoption in OGC Standards:

- REST service encoding: currently WFS, WPS, and CSW only provide RPC or SOAP bindings. The adoption of RESTful bindings will allow the creation of simpler services and potentially increase the adoption of OGC standards.

- The output of WPS as a WFS: to harmonize the passing of data between OGC services it is necessary that the results of a WPS in asynchronous execution mode behave in a way that is indistinguishable from a WFS. Although it renders the WPS slightly more complex, the advantages compensate the complexity.

- JSON format as default for REST: as discussed in this thesis, JSON is now the most used format in APIs. The adoption of JSON as an alternative for XML will potentially increase the use of OGC standards.

- JSON-LD Context + JSON Schema for describing features: JSON-LD extends JSON with the capabilities of adding semantics to attributes and providing linking capabilities similar to XLink. JSON Schema allows the syntactic description and validation of a JSON file, and it can be used as an alternative for XSD.

- Service metadata based on Hydra Core Vocabulary: Hydra allows a description of the service functionality at the resource granularity. Hydra metadata is semantically enabled using a controlled vocabulary and JSON-LD, which allows a direct conversion to RDF. Also by using Hydra, the OGC Web Services can better integrate with other RESTful APIs.

- Extended GeoJSON: regular GeoJSON cannot be directly extended to JSON-LD due to problems in representing lists of lists in the transformation between JSON-LD and RDF. Also, GeoJSON cannot provide semantics at feature type and geometric level. In this thesis, we propose ExtGeoJSON, an extension of GeoJSON that solves both of these problems, making the GeoJSON format more flexible.

- Use of CORS: currently OGC standards do not discuss how one can handle cases of cross-origin data sharing. Some implementations of OWS allow data transfer in the JSON-P format, but this is not desirable, as specific parsers become necessary. In this thesis, we use CORS, which allows the server and the client to specify which data transfer is allowed in a cross-origin request. CORS has the advantage of being implemented in all modern browsers and does not require the transfer of a specific data format.

- Standardization of the HTTP error codes across the specifications: OGC standards do not correctly use HTTP error codes across the defined services. By using HTTP error codes with the correct semantics, generic clients can better handle errors.

- Extension of Simple Feature standard to allow multiple measurement values: currently OGC limits the number of measurement dimensions to one. This limitation reduces the use cases and creates inconsistencies when dealing with other formats such as the GPX specification.

- Units of measurement following the Quantities, Units, Dimensions and Data Types Ontologies (QUDT): currently, in OGC standards, units are specified with a free text entry. By using an ontology for units of measurement, the interoperability of OGC services will increase.

8.4 RECOMMENDATIONS FOR FUTURE WORK

In this section, we identify the aspects of the research that still need further development.

- Consider transactions in the theory of composability: at all levels of composability, the focus was on stateless services, not considering transactions. When performing multiple transactions with different services in a workflow, the atomicity property becomes a challenge and a specialized mechanism to handle this case is needed.

- JSON-based RESTful Bindings for Web Coverage Service (WCS), and Sensor Observation Service (SOS): following the ideas presented in this thesis, one could define RESTful bindings for WCS and SOS. In the case of SOS, one possible reference is the SensorThings API.

- Extend type theory to cover SOS cases: the type theory considers only WPS, WFS, and WCS, but the author expects that the theory is also applicable for SOS. Nevertheless, SOS services need to be further studied for their type-theoretic treatment.

- Semantic Composability: in this thesis, we briefly discussed this level of composability and much work is left to be done. However, in the author's opinion, the use of complex semantics will most likely lead to only inapplicable methods that only make sense in the academic domain. The use of lightweight ontologies and simple relationships will reduce the toolchain and the overall complexity of the application. Also, the composability can become undecidable for complex ontologies, for example, the use of SKOS ontology with relationships such as *closeMatch*, *narrowMatch*, *relatedMatch* will most likely turn the semantic composability to come intractable.

- Qualitative Composability: in this thesis we did not discuss the last level of composability introduced, and a mechanism for its verification is still needed. The concepts of response time, service, availability, cost, security, legal rights, and quality of operations then require formal definitions. Also is needed a notation of the specification of client requirements. This notation ideally should be generated automatically from a natural language description.

- Extension of Filter Encoding JSON serialization to handle multiple resources: in this thesis, we introduce JSFE notation as an alternative to XML Filter Encoding. However, the notation is limited to simple queries against a single resource. It is recommended to define an alternative for XPath in XML that applies to JSON. This should enable complex queries such as spatial and temporal joins.

- Optimization of compositions: several approaches can be used to optimize the execution of workflows, for example: removal of redundant, or unnecessary operations; selection of servers with more processing capabilities; use of the code shipping paradigm to reduce data

transferring; mechanism to reduce the transfer of unnecessary data; workflow rewrite when one service can provide the functionality of multiple operations; load balancing when one service is used several times in a composition.

- Automatic service composition: provide a method to suggest workflows based on user requirements, which allows not only to suggest built workflows (such as in a workflow discovery) but when possible, dynamically build a workflow that fulfills the user requirements.
- Stream processing: when working with SOS, is desirable to have processing services capable of stream processing. This requires adaptation of both processing and data services, creating special cases when performing compositions. This can potentially be implemented using WebSockets.
- Substitutability of services: recomposition at run-time to achieve better performance or to replace unavailable services. In this thesis, we discussed briefly the substitution of individual services but a far more challenging problem is the multi-service substitutability touched upon in Chapter 5.
- Discovery of Workflows: the proposed mechanism of metadata propagation allows the automatic generation of composition metadata, however, most likely this metadata is not sufficient for efficient discovery of workflows. One key point that is missing in the metadata of workflows is semantics. A mechanism that can capture the semantics of the workflow from the semantics of its components seems from the view of the author the biggest challenge in workflow discovery. In Chapter 5, we discussed the concept of semantic propagation with the potential use of SWRL, this approach should be considered.
- Evaluate JSON-W: it is desirable to test and evaluate JSON-W notation in different application domains. One could start by comparing it with SensorML and OpenMI standards.
- Validation of JSON-LD at RDF level: in this thesis, we only perform a JSON Schema validation against JSON files. In the case of JSON-LD, this is not desirable since one can have JSON-LD files with the same contents but following a different structure. In this way, there exist two alternatives: validate the JSON-LD file at the RDF level, or first use JSON-LD framing to bring the file into a known structure and then apply the JSON Schema validation. Both methods should be implemented and compared regarding computational complexity and expressiveness of the validation.
- Verify how HTTP/2 modifies or enhances the ideas presented in this thesis: the HTTP/2 standard provides three main capabilities: improved performance, security, server push. The author expects that HTTP/2 push can be applied instead of WebSockets, and a performance comparison is needed. HTTP/2 will also make REST more efficient by providing means of compressing headers and multiplexing, which allows multiple HTTP requests to run in parallel and also allows multiple responses to be sent with only a single request. HTTP/2 also migrates from a textual protocol to a binary one, which has a lower overhead to transmit, and is less error-prone.
- Integration with other types of services: The proposed solution is only applicable to JSON-LD-based, RESTful, Hydra-enabled Web Services and considers only the asynchronous execution mode of WPS. One needs to verify how those very specific services can be integrated with the WS-* family of services, which are XML- and SOAP-based.

LIST OF REFERENCES

[1] S. Donkervoort, S. Dolan, M. Beckwith, T. Northrup, and A. Sozer, "Enhancing accurate data collection in mass fatality kinship identifications: Lessons learned from Hurricane Katrina," *Forensic Sci. Int. Genet.*, no. 2, pp. 354–362, 2008.

[2] D. Hintz, "GIS for Disaster Response," 2007. [Online]. Available: http://archives.profsurv.com/magazine/article.aspx?i=70147#Disaster_Response

[3] N. Wiegand and C. Garcia, "A task-based ontology approach to automate goespatial data retrieval," *Transactions in GIS*, no. 11, pp. 355–376, 2007.

[4] NASA, "A.40 computational modeling algorithms and cyberinfrastructure," National Aeronautics and Space Administration (NASA), Tech. Rep., 2012.

[5] W. Yaxing, S. Santhana-Vannan, and R. Cook, "Discover, visualize, and deliver geospatial data through OGC standards-based WebGIS system," in *Proc. 17th Internation Conference on Geoinformatics*, Fairfax, VA, 2009, pp. 1–6.

[6] Open Geospatial Consortium Inc, "OpenGIS Web Feature Service 2.0 Interface Standard," Open Geospatial Consortium Inc, Tech. Rep., 2010. [Online]. Available: http://www.opengeospatial.org/standards/wfs

[7] Open Geospatial Consortium Inc, "OGC WPS 2.0 Interface Standard," Open Geospatial Consortium Inc, Tech. Rep., 2015. [Online]. Available: http://docs.opengeospatial.org/is/14-065/14-065.html

[8] Open Geospatial Consortium Inc, "OpenGIS Catalogue Services Specification," Open Geospatial Consortium Inc, Tech. Rep., 2007. [Online]. Available: http://www.opengeospatial.org/standards/cat

[9] Open Geospatial Consortium Inc, "Geospatial Semantic Web Interoperability Experiment," Open Geospatial Consortium Inc, Tech. Rep., 2006. [Online]. Available: http://www.opengeospatial.org/projects/initiatives/gswie

[10] W. Kuhn, "Geospatial semantics: why, of what, and how?" *Journal on Data Semantics III*, no. 3, pp. 1–24, 2005.

[11] T. Berners-Lee, "Linked Data - Design Issues," 2009. [Online]. Available: http://www.w3.org/DesignIssues/LinkedData.html

[12] G. Hart and C. Dolbear, *Linked Data: a Geographic Perspective*. Boca Raton, FL: CRC Press, 2013. [Online]. Available: http://ezproxy.utwente.nl:2100/isbn/978-1-4398-6995-6

[13] Google, "Google Trends - Web Search interest: Semantic Web - Worldwide, 2004 - present," 2015. [Online]. Available: https://www.google.com/trends/explore#q=%2Fm%2F076k0

[14] TopQuadrant Inc, "TopQuadrant," 2015. [Online]. Available: http://www.topquadrant.com/

[15] Thomson Reuters |, "Open Calais," 2015. [Online]. Available: http://new.opencalais.com/

[16] IBM, "IBM Watson," 2015. [Online]. Available: http://www.ibm.com/smarterplanet/us/en/ibmwatson/

[17] Open Geospatial Consortium Inc, "OGC GeoSPARQL - A geographic query language for RDF data," Open Geospatial Consortium Inc, Tech. Rep., 2012. [Online]. Available: http://www.opengeospatial.org/standards/geosparqlhttp://www.opengis.net/doc/IS/geosparql/1.0

[18] Google, "Google Trends," 2015. [Online]. Available: https://www.google.com/trends/

[19] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[20] Object Management Group, "Business Process Model and Notation (BPMN) Version 2.0," Object Management Group, Tech. Rep., 2011.

[21] Open Geospatial Consortium Inc, "OGC Reference Model," Open Geospatial Consortium Inc, Tech. Rep., 2011.

[22] ProgrammableWeb, "ProgrammableWeb - APIs, Mashups and the Web as Platform," 2016. [Online]. Available: http://www.programmableweb.com/

[23] Google Code blog, "A well earned retirement for the SOAP Search API," 2009. [Online]. Available: http://googlecode.blogspot.nl/2009/08/well-earned-retirement-for-soap-search.html

[24] Open Geospatial Consortium Inc, "Testbed 11 Implementing JSON/GeoJSON in an OGC Standard Engineering Report," Open Geospatial Consortium Inc, Tech. Rep., 2015.

[25] Open Geospatial Consortium Inc, "Testbed 10 Rules for JSON and GeoJSON Adoption: Focus on OWS-Context," Open Geospatial Consortium Inc, Tech. Rep., 2014.

[26] Open Geospatial Consortium Inc, "The OGC Forms REST and WFS/FE Standards Working Groups," 2011. [Online]. Available: http://www.opengeospatial.org/node/1497

[27] Open Geospatial Consortium Inc, "OGC Web Map Tile Service Implementation Standard," Open Geospatial Consortium Inc, Tech. Rep., 2010.

[28] Open Geospatial Consortium Inc, "OGC RESTful encoding of OGC Sensor Planning Service for Earth Observation satellite Tasking," Open Geospatial Consortium Inc, Tech. Rep., 2014.

[29] Open Geospatial Consortium Inc, "OGC RESTful Encoding of Ordering Services Framework For Earth Observation Products," Open Geospatial Consortium Inc, Tech. Rep., 2014.

[30] Open Geospatial Consortium Inc, "OGC Download Service for Earth Observation Products," Open Geospatial Consortium Inc, Tech. Rep., 2014.

[31] Open Geospatial Consortium Inc, "OGC SensorThings API," Open Geospatial Consortium Inc, Tech. Rep., 2015. [Online]. Available: http://ogc-iot.github.io/ogc-iot-api/index.html

[32] Esri, "GeoServices REST Specification," Esri, Tech. Rep. September, 2010.

[33] C. Shorter, "GeoServices REST API adoption vote - update and motion to withdraw," 2013. [Online]. Available: http://lists.osgeo.org/pipermail/discuss/2013-May/011789.html

[34] P. Panagiotis, "A REST binding for WFS 2.0," p. 3, 2011. [Online]. Available: https://portal.opengeospatial.org/files/?artifact_id=44852

[35] Open Geospatial Consortium Inc, "Testbed 11 REST Interface Engineering Report," Open Geospatial Consortium Inc, Tech. Rep., 2015.

[36] C. Granell, L. Díaz, A. Tamayo, and J. Huerta, "Assessment of OGC Web Processing Services for REST principles," *Data Mining, Modelling and Management*, 2012. [Online]. Available: http://arxiv.org/abs/1202.0723

[37] T. Foerster, A. Brühl, and B. Schäffer, "RESTful Web Processing Service," in *14th AGILE International Conference on Geographic Information Science*, 2011. [Online]. Available: http://ifgi.uni-muenster.de/$\sim$tfoer_01/articles/AGILE2011_Foerster_etal_RESTful_WPS.pdf

[38] L. C. Jiang, P. Yue, and X. C. Lu, "Restful Implementation of Catalogue Service for Geospatial Data Provenance," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XL-4/W2, no. November, pp. 121–125, 2013. [Online]. Available: http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XL-4-W2/121/2013/

[39] P. Mazzetti, S. Nativi, and J. Caron, "RESTful implementation of geospatial services for Earth and Space Science applications," *International Journal of Digital Earth*, vol. 2, no. sup1, pp. 40–61, 2009.

[40] K. Janowicz, A. Bröring, C. Stasch, S. Schade, T. Everding, and A. Llaves, "A RESTful proxy and data model for linked sensor data," *International Journal of Digital Earth*, pp. 1–22, 2011.

[41] K. R. Page, D. C. D. Roure, K. Martinez, J. D. Sadler, and O. Y. Kit, "Linked Sensor Data: RESTfully serving RDF and GML," in *Semantic Sensor Networks*, 2009, pp. 49–63. [Online]. Available: http://eprints.ecs.soton.ac.uk/21743/1/ssn09-krp-03.pdf

[42] 52 North Initiative for Geospatial Open Source Software GmbH, "SOS RESTful Add-on," 2015. [Online]. Available: http://52north.org/communities/sensorweb/sosREST/

[43] 52 North Initiative for Geospatial Open Source Software GmbH, "Sensor Web REST API," 2015. [Online]. Available: http://sensorweb.demo.52north.org/sensorwebclient-webapp-stable/api-doc/

[44] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, 2000. [Online]. Available: https://www.ics.uci.edu/$\sim$fielding/pubs/dissertation/top.htm

[45] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231

[46] Oasis, "OData - Open Data Protocol," 2015. [Online]. Available: http://www.odata.org/

[47] N. Freed and N. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types,” 1996. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2046.txt

[48] Oasis, “Batch Processing (OData Version 3.0),” 2015. [Online]. Available: http://www.odata.org/documentation/odata-version-3-0/batch-processing/

[49] M. Muller, L. Bernard, and J. Brauner, “Moving Code in Spatial Data Infrastructures - Web Service Based Deployment of Geoprocessing Algorithms,” *Transactions in GIS*, 2010.

[50] Open Geospatial Consortium Inc, “OGC Filter Encoding 2.0 Encoding Standard With Corrigendum,” Open Geospatial Consortium Inc, Tech. Rep., 2014.

[51] I. Fette and A. Melnikov, “The WebSocket Protocol,” 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455

[52] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” 2014. [Online]. Available: https://tools.ietf.org/html/rfc7159

[53] M. Lanthaler and C. Gütl, “On Using JSON-LD to Create Evolvable RESTful Services,” in *WS-REST ’12*, 2012, pp. 25–32.

[54] R. Lämmel and E. Meijer, “Revealing the X/O impedance mismatch,” *Datatype-Generic Programming*, pp. 1–80, 2007. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-76786-2_6

[55] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and C. Schmidt, “GeoJSON,” 2008. [Online]. Available: http://geojson.org/

[56] Open Geospatial Consortium Inc, “OGC OWS Context GeoJSON Encoding Standard,” Open Geospatial Consortium Inc, Tech. Rep., 2015.

[57] Open Geospatial Consortium, “Opengis implementation standard for geographic information - Simple feature access - Part 1: Common Architecture,” Open Geospatial Consortium Inc, Tech. Rep., 2011. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=25355E+Implementation+Standard+for+Geographic+information+-+Simple+feature+access#1

[58] MongoDB Inc, “MongoDB,” 2016. [Online]. Available: https://www.mongodb.org/

[59] F. Galiegue, K. Zyp, and G. Court, “JSON Schema,” 2013. [Online]. Available: http://json-schema.org/

[60] F. Galiegue, “Sample JSON schemas,” 2014. [Online]. Available: https://github.com/fge/sample-json-schemas/

[61] International Organization for Standardization, “ISO 19119:2005 Geographic Information - Services,” International Organization for Standardization, Geneva, Tech. Rep., 2005.

[62] R. d. S. Rocha, L. C. Degrossi, and J. P. de Albuquerque, “A Systematic Literature Review of Geospatial Web Service Composition,” in *XVIII Ibero-American Conference on Software Engineering*, 2015. [Online]. Available: http://www.agora.icmc.usp.br/site/wp-content/uploads/2015/03/Rocha_ESELAW_2015.pdf

[63] H. Rusli, S. Ibrahim, and M. Puteh, "Testing Web Services Composition: A Mapping Study," *Communications of the IBIMA*, pp. 1–12, 2011. [Online]. Available: http://www.ibimapublishing.com/journals/CIBIMA/2011/598357/598357.html

[64] N. Milanovic, "Contract-based web service composition framework with correctness guarantees," in *Second International Service Availability Symposium, ISAS*, vol. 3694 LNCS, 2005, pp. 52–67.

[65] G. Baryannis, M. Carro, and D. Plexousakis, "Deriving Specifications for Composite Web Services," in *2012 IEEE 36th International Conference on Computer Software and Applicatiosn*, 2012, pp. 432–437.

[66] Z. Chen, J. Wu, S. Deng, Y. Li, and Z. Wu, "Describing and Verifying Web Service Using Type Theory," in *10th International Conference on Computer Supported Cooperative Work in Design*, 2006, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4019219

[67] A. Kim, E. Ioup, M. Kang, C. Meadows, and J. Sample, "A Framework for Automatic Web Service Composition," Naval Research Laboratory, Tech. Rep., 2009. [Online]. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a499917.pdf

[68] B. Medjahed and A. Bouguettaya, "A multilevel composability model for semantic Web services," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 7, pp. 954–968, 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1432704

[69] R. Tarjan, "Depth-first search and linear graph algorithms," *12th Annual Symposium on Switching and Automata Theory*, vol. 1, no. 2, pp. 146–160, 1971.

[70] M. Sipser, "The Halting Problem," in *Introduction to the Theory of Computation*, 2nd ed. Course Technology, 2006.

[71] L. Cardelli, "Type Systems," in *CRC Handbook of Computer Science and Engineering*, 2nd ed., A. B. Tucker, Ed. Chapman and Hall/CRC, 2004, ch. 97, pp. 2208–2236. [Online]. Available: http://lucacardelli.name/papers/typesystems.pdf

[72] International Organization for Standardization, "ISO 19125-1:2004 Geographic information - Simple feature access - Part 1: Common architecture," International Organization for Standardization, Geneva, Tech. Rep., 2004.

[73] Open Geospatial Consortium Inc, "A Conceptual Model and Text-based Notation for Temporal Geometry," Open Geospatial Consortium Inc, Tech. Rep., 2014. [Online]. Available: http://www.opengeospatial.org/blog/2357

[74] International Organization for Standardization, "ISO 8601 - Data elements and interchange formats - Information interchange - Representation of dates and times," International Organization for Standardization, Geneva, Tech. Rep., 2004.

[75] International Organization for Standardization, "ISO 19108:2002 Geographic information - Temporal schema," International Organization for Standardization, Geneva, Tech. Rep., 2002.

[76] Open Geospatial Consortium Inc, "OpenGIS Abstract Specification Topic 6: Schema for coverage geometry and functions," Open Geospatial Consortium Inc, Tech. Rep., 2006.

[77] Open Geospatial Consortium Inc, "OGC GML Application Schema - Coverages," Open Geospatial Consortium Inc, Tech. Rep., 2010.

[78] International Organization for Standardization, "ISO 19123:2005 Geographic information - Schema for coverage geometry and functions," International Organization for Standardization, Geneva, Tech. Rep., 2005.

[79] International Organization for Standardization, "ISO 19101-1:2005 Geographic information - Reference model - Part 1: Fundamentals," International Organization for Standardization, Geneva, Tech. Rep., 2005.

[80] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–523, 1985.

[81] PostGIS, "PostGIS 2.2.2dev Manual," 2015. [Online]. Available: http://postgis.net/docs/manual-2.2/

[82] R. Hodgson, P. J. Keller, J. Hodges, and J. Spivak, "QUDT - Quantities, Units, Dimensions and Types," 2014. [Online]. Available: http://www.qudt.org/

[83] Open Geospatial Consortium Inc, "OGC Name Type Specification for Coordinate Reference Systems," Open Geospatial Consortium Inc, Tech. Rep., 2013.

[84] Garmin, "GPX TrackPointExtension v1," 2004. [Online]. Available: https://www8.garmin.com/xmlschemas/TrackPointExtensionv1.xsd

[85] Open Geospatial Consortium Inc, "Geographic information - Well-known text representation of coordinate reference systems," Open Geospatial Consortium Inc, Tech. Rep., may 2015. [Online]. Available: http://docs.opengeospatial.org/is/12-063r5/12-063r5.html

[86] M. Hedley and C. Little, "Advancing our ability to tell time," 2016. [Online]. Available: http://www.opengeospatial.org/blog/2357

[87] IEEE, "Seconds Since the Epoch," 2013. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15

[88] Safe Software, "Rubber Sheeter | FME by Safe Software," 2016. [Online]. Available: https://www.safe.com/transformers/rubber-sheeter/

[89] T. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *International Journal of Human-Computer Studies*, vol. 43, no. 5-6, pp. 907–928, 1995. [Online]. Available: http://dx.doi.org/10.1006/ijhc.1995.1081

[90] W3C Working Group, "OWL 2 Web Ontology Language Document Overview (Second Edition)," 2012. [Online]. Available: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/

[91] W3C, "RDF 1.1 Concepts and Abstract Syntax," 2014. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[92] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers, "RDF 1.1 Turtle," 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-turtle-20140225/

[93] J. Brauner, "Formalizations for geooperators - Geoprocessing in spatial data infrastructures," PhD, Technische Universitat Dresden, 2015.

[94] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," 2004. [Online]. Available: http://www.w3.org/Submission/SWRL/

[95] K. Janowicz, F. V. Harmelen, J. a. Hendler, and P. Hitzler, "Why the data train needs semantic rails," *AI Magazine, Association for the Advancement of Artificial Intelligence (AAAI)*, no. May, pp. 1–8, 2014.

[96] W. Kuhn, T. Kauppinen, and K. Janowicz, "Linked Data - A Paradigm Shift for Geographic Information Science," in *8th International Conference, GIScience 2014*, 2014.

[97] Open Geospatial Consortium Inc, "Testbed 11 Implementing Linked Data and Semantically Enabling OGC Services Engineering Report," Open Geospatial Consortium Inc, Tech. Rep., 2015.

[98] K. Janowicz, S. Scheider, T. Pehle, and G. Hart, "Geospatial semantics and linked spatiotemporal data - past, present, and future," *Semantic Web*, vol. 3, no. 4z, pp. 321–332, 2012.

[99] F. Harvey, J. Jones, S. Scheider, A. Iwaniak, I. Kaczmarek, J. Lukowicz, and M. Strzelecki, "Little Steps Towards Big Goals. Using Linked Data to Develop Next Generation Spatial Data Infrastructures (aka SDI 3.0)," in *Proceedings of the AGILE 2014 - International Conference on Geographic Information Science*, 2014, pp. 3–6.

[100] C. Zhang, T. Zhao, and W. Li, "Automatic search of geospatial features for disaster and emergency management," *International Journal of Applied Earth Observation and Geoinformation*, vol. 12, no. 6, pp. 409–418, 2010.

[101] T. Zhao, C. Zhang, M. Wei, and Z. R. Peng, "Ontology-based geospatial data query and integration," in *5th International Conference, GIScience 2008*, vol. 5266 LNCS, no. 1, Park City, UT, USA, 2008, pp. 370–392.

[102] J. Jones, W. Kuhn, C. Kebler, and S. Scheider, "Making the Web of Data Available Via Web Feature Services," in *Connecting a Digital Europe through Location and Place. Proceedings of the AGILE'2014 International Conference on Geographic Information Science*. Springer International Publishing, 2014, ch. Part V, pp. 341–361. [Online]. Available: http://link.springer.com/10.1007/978-3-319-03611-3

[103] K. Janowicz, S. Schade, A. Bröring, C. Keßler, P. Maué, and C. Stasch, "Semantic Enablement for Spatial Data Infrastructures," *Transactions in GIS*, vol. 14, no. 2, pp. 111–129, 2010.

[104] P. Bryan and K. Zyp, "JSON Reference," 2013. [Online]. Available: https://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03

[105] M. Kelly, "The Hypertext Application Language," 2013. [Online]. Available: http://stateless.co/hal_specification.html

[106] M. Amundsen, "Collection+JSON - Document Format : Media Types," 2013. [Online]. Available: http://amundsen.com/media-types/collection/format/

[107] M. Lanthaler, "A Semantic Description Language for RESTful Data Services to Combat Semaphobia," in *5th IEEE International Conference on Digital Ecosystems and Technologies*, vol. 5, no. June, 2011, pp. 47–53.

[108] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "JSON-LD 1.0," 2014. [Online]. Available: https://www.w3.org/TR/json-ld/

[109] Open Geospatial Consortium Inc, "Geospatial Microtheories," 2015. [Online]. Available: http://ows.usersmarts.com/geospatial/

[110] P. Colpaert, "Coordinates to RDF doesn't work - 32," 2015. [Online]. Available: https://github.com/geojson/geojson-ld/issues/32

[111] S. Gillies, "GeoJSON-LD," 2015. [Online]. Available: https://github.com/geojson/geojson-ld

[112] E. Wilde, "GeoJSON-X," 2015. [Online]. Available: https://github.com/dret/GeoJSON-X

[113] GeOps, "GeoServer WFS Performance Comparison," 2013. [Online]. Available: http://geops.de/blog/130-geoserver-wfs-performance-comparison?language=en

[114] I. Tihonov, "GeoJSON-minifier," 2015. [Online]. Available: https://github.com/igorti/geojson-minifier

[115] Mapbox, "Geobuf," 2015. [Online]. Available: https://github.com/mapbox/geobuf

[116] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," 2007. [Online]. Available: https://www.w3.org/TR/wsdl20/

[117] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "OWL-S: Semantic Markup for Web Services," 2004. [Online]. Available: https://www.w3.org/Submission/OWL-S/

[118] J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, U. Keller, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, H. Lausen, E. Oren, A. Polleres, D. Roman, J. Scicluna, and M. Stollberg, "Web Service Modeling Ontology (WSMO)," 2005. [Online]. Available: https://www.w3.org/Submission/WSMO/

[119] R. Verborgh, T. Steiner, R. V. de Walle, and J. Gabarro, "Linked Data and Linked APIs: Similarities, Differences, and Challenges," in *The Semantic Web: ESWC 2012 Satellite Events*, vol. 7540, 2012, pp. 73–86. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84942616762&partnerID=tZOtx3y1

[120] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML microformat for describing RESTful Web services," in *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2008, pp. 619–625.

[121] J. Lathem, "SA-REST and (S) mashups : Adding Semantics to RESTful Services," in *International Conference on Semantic Computing*, 2007, pp. 469–476.

[122] M. Lanthaler, "Creating 3rd Generation Web APIs with Hydra," in *WWW 2013 Companion*, 2013, pp. 2–4.

[123] Multimedia Lab Ghent University - iMind, "Linked Data Fragments," 2016. [Online]. Available: http://linkeddatafragments.org/

[124] R. Verborgh, "600,000 queryable datasets - and counting," 2015. [Online]. Available: http://ruben.verborgh.org/blog/2015/01/30/600000-queryable-datasets-and-counting/

[125] M. Lanthaler, "Hydra Core Vocabulary," 2015. [Online]. Available: http://www.hydra-cg.com/spec/latest/core/

[126] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. Den Bussche, "The Open Provenance Model core specification (v1.1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.future.2010.07.005

[127] W3C Working Group, "PROV-O: The PROV Ontology," 2013. [Online]. Available: https://www.w3.org/TR/prov-o/

[128] S. A. Cook, "A taxonomy of problems with fast parallel algorithms," *Information and Control*, vol. 64, no. 1-3, pp. 2–22, jan 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995885800413

[129] A. Bucchiarone and S. Gnesi, "A Survey on Services Composition Languages and Models," in *International Workshop on Web Services Modelling and Testing*, 2006, pp. 1–13.

[130] P. Wohed, W. Aalst, M. Dumas, and A. Hofstede, "Analysis of Web Services Composition Languages: The Case of BPEL4WS," in *22nd International Conference on Conceptual Modeling*, vol. 2813, Chicago, IL, USA, 2003, pp. 200–215.

[131] W. M. P. Van der Aalst, a. H. M. Ter Hofstede, B. Kiepuszewski, and a. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, pp. 5–51, 2003.

[132] W. van der Aalst and A. ter Hofstede, "Workflow Patterns Home Page," 2011. [Online]. Available: http://www.workflowpatterns.com/

[133] Open Geospatial Consortium Inc, "SensorML: Model and XML Encoding Standard," Open Geospatial Consortium Inc, Tech. Rep., 2014.

[134] OpenMI Association, "OpenMI," 2015. [Online]. Available: http://www.openmi.org/

[135] Node.js Foundation, "Node.js," 2016. [Online]. Available: https://nodejs.org/en/

[136] Mapbox, "Turf.js," 2016. [Online]. Available: http://turfjs.org/

[137] B. Harrtell, "JSTS Topology Suite," 2016. [Online]. Available: https://github.com/bjornharrtell/jsts

[138] Natural Atlas, "node-gdal," 2016. [Online]. Available: https://github.com/naturalatlas/node-gdal

[139] LearnBoost, "Mongoose ODM v4.3.7," 2016. [Online]. Available: http://mongoosejs.com/

[140] Socket.IO, "Socket.IO," 2016. [Online]. Available: http://socket.io/

[141] Getify Solutions, "JSON-P: Safer cross-domain Ajax with JSON-P/JSONP," 2010. [Online]. Available: http://json-p.org/

[142] M. Hossain and M. Hausenblas, "Enable cross-origin resource sharing," 2016. [Online]. Available: http://enable-cors.org/

[143] G. Luff, "Tiny Validator (for v4 JSON Schema)," 2016. [Online]. Available: https://github.com/geraintluff/tv4

[144] C. McMahon, "Async.js," 2016. [Online]. Available: https://github.com/caolan/async

[145] A. ONeal, "node-uuid," 2015. [Online]. Available: https://github.com/broofa/node-uuid

[146] P. Leach, M. Mealling, and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace," 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4122.txt

[147] Google, "AngularJS - Superheroic JavaScript MVW Framework," 2016. [Online]. Available: https://angularjs.org/

[148] V. Agafonkin, "Leaflet - a JavaScript library for interactive maps," 2016. [Online]. Available: http://leafletjs.com/

[149] I. S. Ortega, "Leaflet.draw," 2015. [Online]. Available: https://github.com/Leaflet/Leaflet.draw

[150] S. S. Walia, A. Dasgupta, and S. K. Ghosh, "Geospatial orchestration framework for resolving complex user query," in *ICCSA 2011*, vol. 6782 LNCS, no. PART 1, 2011, pp. 643–651.

[151] X. Meng, Y. Xie, and F. Bian, "Distributed geospatial analysis through web processing service: A case study of earthquake disaster assessment," *Journal of Software*, vol. 5, no. 6, pp. 671–679, 2010.

[152] J. de Jesus, P. Walker, M. Grant, and S. Groom, "WPS orchestration using the Taverna workbench: The eScience approach," *Computers and Geosciences*, vol. 47, pp. 75–86, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.cageo.2011.11.011

[153] A. Akram, D. Meredith, and R. Allan, "Evaluation of BPEL to Scientific Workflows," in *Sixth IEEE International Symposium on Cluster Computing and the Grid*, 2006, pp. 4–7.

[154] Open Geospatial Consortium Inc, "Workflow DWG | OGC," 2016. [Online]. Available: http://www.opengeospatial.org/projects/groups/workflowdwg

Appendix A

JSON Serialization for Filter Encoding

Filter Encoding is necessary for the definition of Catalogue Services for the Web (CSW) and optional in Web Feature Services (WFS). It is standardized in OGC Filter Encoding 2.0 Standard and ISO 19143 – Filter Encoding. In this appendix, we exemplify operations of JSON Serialization for Filter Encoding (JSFE). All operations are based on MongoDB query parameters. It uses GeoJSON geometry encoding, in contrast with OGC Filter Encoding that uses GML geometries.

Table A.1 presents the syntax for JSFE in Backus-Naur Form (BNF). Table A.2 presents a comparison between XML Filter Encoding and JSFE.

Table A.1: BNF for JSFE

<expression> ::= {$and: [<expression>,...,<expression>]}
| {$or: [<expression>,...,<expression>]}
| {$not: <expression>}
| <standard filter>
| <spatial filter>

<standard filter> ::= {<attribute>: {$<simple operator>: <numeric>}}
| {<attribute>: {$exists: *true*}}
| {<attribute>: {$exists: *false*}}
| {<attribute>: {$between: [<numeric>, <numeric>]}}
| {<attribute>: {$regex: <string>, $wildCard: <char>,
$singleChar: <char>, $escape: <char>}}

<simple operator> ::= lt|lte|gt|gte|eq

<spatial filter> ::= {<attribute>: {$<simple spatial operator>: <geometry>}}
| {<attribute>: {$<binary spatial operator>: <geometry>,
$distance: <numeric>}}}
| {geom: {$bbox: <bbox>}}}

<simple spatial operator> ::= equal|disjoint|touches|within|overlaps|crosses|intersects|contains

<binary spatial operator> ::= within|beyond

Table A.2: JSFE operators

Operator	Filter Encoding	JSFE
Property Is Null	<PropertyIsNull> <PropertyName>*name*</PropertyName> </PropertyIsNull>	{name: {$exists: *true*}}
Property Is Nil	<PropertyIsNil> <PropertyName>*name*</PropertyName> </PropertyIsNil>	{name: {$exists: *false*}}
Property Is Less Than	<PropertyIsLessThan> <PropertyName>*age*</PropertyName> <Literal>18</Literal> </PropertyIsLessThan>	{age: {$lt: 18}}
Property Is Less Than Or Equal To	<PropertyIsLessThanOrEqualTo> <PropertyName>*age*</PropertyName> <Literal>18</Literal> </PropertyIsLessThanOrEqualTo>	{age: {$lte: 18}}
Property Is Greater Than	<PropertyIsGreaterThan> <PropertyName>*age*</PropertyName> <Literal>18</Literal> </PropertyIsGreaterThan>	{age: {$gt: 18}}
Property Is Greater Than Or Equal To	<PropertyIsGreaterThanOrEqualTo> <PropertyName>*age*</PropertyName> <Literal>18</Literal> </PropertyIsGreaterThanOrEqualTo>	{age: {$gte: 18}}
Property Is Equal To	<PropertyIsEqualTo> <PropertyName>*age*</PropertyName> <Literal>18</Literal> </PropertyIsEqualTo>	{age: {$eq: 18}}
Property Is Between	<PropertyIsBetween> <PropertyName>*age*</PropertyName> <LowerBoundary>18</LowerBoundary> <UpperBoundary>60< /UpperBoundary> </PropertyIsBetween>	{age: {$between: [18, 60]}}
Property Is Like	<PropertyIsLike wildCard="*" singleChar="." escape="!"> <PropertyName>*owmstat*</PropertyName> <Literal>*Natuurlij**</Literal> </PropertyIsLike>	{owmstat: {$regex: *"Natuurlij**"*, $wildCard: *"**"*, $singleChar: *"."*, $escape: *"!"*}}
And	<And> ... </And>	{$and: [...]}
Or	<Or> ... </Or>	{$or: [...]}
Not	<Not> ... </Not>	{$not: {...}}
Equal	<Equal> <PropertyName>*geom*</PropertyName> <gml:Point> <gml:coordinates>125.6 10.1</gml:coordinates> </gml:Point> </Equal>	{geom: {$equal: {geometry: {type: *"Point"*, coordinates: [125.6, 10.1]}}}}

Operator	Filter Encoding	JSFE
Disjoint	<Disjoint> <PropertyName>*geom*</PropertyName> <gml:Point> <gml:coordinates>125.6 10.1</gml:coordinates> </gml:Point> </Disjoint>	{geom: {$disjoint: {geometry: {type: *"Point"*, coordinates: [125.6, 10.1]}}}}
Touches	<Touches> <PropertyName>*geom*</PropertyName> <gml:Point> <gml:coordinates>125.6 10.1</gml:coordinates> </gml:Point> </Touches>	{geom: {$touches: {geometry: {type: *"Point"*, coordinates: [125.6, 10.1]}}}}
Within	<Within> <PropertyName>*geom*</PropertyName> <gml:Polygon> <gml:outerBoundaryIs> <gml:LinearRing> <gml:coordinates>-10.0 -10.0, 10.0 -10.0, 10.0 10.0, -10.0, 10.0</gml:coordinates> </gml:LinearRing> </gml:outerBoundaryIs> </gml:Polygon> </Within>	{geom: {$within: {geometry: {type: *"Polygon"*, coordinates: [[[-10.0, -10.0], [10.0, -10.0], [10.0, 10.0], [-10.0, 10.0]]]}}}}
Overlaps	<Overlaps> <PropertyName>*geom*</PropertyName> <gml:Polygon> <gml:outerBoundaryIs> <gml:LinearRing> <gml:coordinates>-10.0 -10.0, 10.0 -10.0, 10.0 10.0, -10.0, 10.0</gml:coordinates> </gml:LinearRing> </gml:outerBoundaryIs> </gml:Polygon> </Overlaps>	{geom: {$overlaps: {geometry: {type: *"Polygon"*, coordinates: [[[-10.0, -10.0], [10.0, -10.0], [10.0, 10.0], [-10.0, 10.0]]]}}}}
Crosses	<Crosses> <PropertyName>*geom*</PropertyName> <gml:Polygon> <gml:outerBoundaryIs> <gml:LinearRing> <gml:coordinates>-10.0 -10.0, 10.0 -10.0, 10.0 10.0, -10.0, 10.0</gml:coordinates> </gml:LinearRing> </gml:outerBoundaryIs> </gml:Polygon> </Crosses>	{geom: {$crosses: {geometry: {type: *"Polygon"*, coordinates: [[[-10.0, -10.0], [10.0, -10.0], [10.0, 10.0], [-10.0, 10.0]]]}}}}
Intersects	<Intersects> <PropertyName>*geom*</PropertyName> <gml:Point> <gml:coordinates>125.6 10.1</gml:coordinates> </gml:Point> </Intersects>	{geom: {$intersects: {geometry: {type: *"Point"*, coordinates: [125.6, 10.1]}}}}

Operator	Filter Encoding	JSFE
Contains	<Contains> <PropertyName>*geom*</PropertyName> <gml:Point> <gml:coordinates>125.6 10.1</gml:coordinates> </gml:Point> </Contains>	{geom: {$contains: {geometry: {type: *"Point"*, coordinates: [125.6, 10.1]}}}}
DWithin	<DWithin> <PropertyName>*geom*</PropertyName> <gml:Point> <gml:coordinates>125.6 10.1</gml:coordinates> </gml:Point> <Distance>100</Distance> </Dwithin>	{geom: {$dWithin: {geometry: {type: *"Point"*, coordinates: [125.6, 10.1]}, $distance: 100}}}
Beyond	<Beyond> <PropertyName>*geom*</PropertyName> <gml:Point> <gml:coordinates>125.6 10.1</gml:coordinates> </gml:Point> <Distance>100</Distance> </Beyond>	{geom: {$beyond: {geometry: {type: *"Point"*, coordinates: [125.6, 10.1]}, $distance: 100}}}
BBOX	<BBOX> <PropertyName>*geom*</PropertyName> <gml:Box><gml:coord> <gml:X>-10</gml:X><gml:Y>0</gml:Y> </gml:coord><gml:coord> <gml:X>10</gml:X><gml:Y>10</gml:Y> </gml:coord></gml:Box></BBOX>	{geom: {$bbox: [-10, 0, 10, 10]}}}

Appendix B

Algorithms for Structural Composability

The Algorithm 1 performs a recursive call of the Algorithms 2 and 3.

Algorithm 1: Branching algorithm for deriving all possible graph scenarios

```
1  Algorithm branchingGraph(G)
       Input: A topologically sorted graph G, with nodes N and the edges E
       Output: a set S of graph scenarios
2      S ← []
3      v ← null
4      forall nodes n in N do
5          if type(n) = conditional then
6              v ← n
7              break
8          end
9      end
10     if v ≠ null then
11         N_marked, E_marked = markGraph(N, E, v)
12         S_0 ← cleanGraph(N_marked, E_marked, v, 0)
13         S_1 ← cleanGraph(N_marked, E_marked, v, 1)
14         S.concatenate(branchingGraph(S_0), branchingGraph(S_1))
15     else
16         S.push((N, E))
17     end
18     return S
```

Algorithm 2: Marking algorithm for a particular conditional

```
1  Algorithm markGraph(N, E, v)
       Input: A graph (N,E) and v a conditional node
       Output: the graph N, E marked
2      S ← []
3      forall edges e from v to w in (N,E).adjacentEdges(v)[false] do
4          S.push(w)                    // initial nodes for depth-first traverse
5          E[e].mark = 0      // select the edge e in E and set the mark value
6      end
7      while S is not empty do
8          u = S.pop()
9          if u.mark = null then
10             N[u].mark = 0      // selected the node u in N and set the mark
11             forall edges e from u to w in (N,E).adjacentEdges(u) do
12                 E[e].mark = 0
13                 S.push(w)
14             end
15         end
16     end
17     forall edges e from v to w in (N,E).adjacentEdges(v)[true] do
18         S.push(w)
19         E[e].mark = 1
20     end
21     while S is not empty do
22         u = S.pop()
23         if u.mark = null || u.mark = 0 then
24             if u.mark = 0 then  /* if the node is marked it receives mark 2 */
25                 N(u).mark = 2
26             else
27                 N(u).mark = 1
28             end
29             forall edges e from u to w in (N,E).adjacentEdges(u) do
30                 if e.mark = 0 then
31                     E(e).mark = 2
32                 else
33                     E(e).mark = 1
34                 end
35                 S.push(w)
36             end
37         end
38     end
39     return (N, E)
```

Algorithm 3: Cleaning algorithm for deriving a particular scenario

```
1  Algorithm cleanGraph(N, E, v, t)
       Input: The marked nodes N and edges E of the graph representation, an conditional
              node v, and the identifier t of the mark (0 or 1)
       Output: the graph N, E cleaned
2      S ← []
3      forall nodes n in N do
4          if n.mark = t then
5              S ← []
6              S.push(n)
7              while S is not empty do
8                  u = S.pop()
9                  forall edges e from w to u in (N,E).adjacentEdges(u) do
10                     if w.mark = t || w.mark = null then
11                         E[e].pop()
12                         S.push(w)
13                     end
14                 end
15                 N[u].pop()
16             end
17         end
18     end
19     initNode ← (N, E).parentNode(v)
20     forall edges e from v to w in (N,E).adjacentEdges(v)[¬ t] do
21         E[e].origin = initNode
22     end
23     return (N, E)
```

Appendix C

JSON Serialization for Types and Conditions

Table C.1 presents the syntax for the JSON serialization of type declarations in Backus-Naur Form (BNF). Table C.2 presents the systax for the JSON serialization pre- and postconditon.

Table C.1: BNF for type declarations

```
                <type> ::= {$set: { <type> }}
                         | {$record: { <name>:<type>,...,<name>:<type> }}
                         | {$union: [<type>,...,<type>]}
                         | <type operator>
                         | <basic type>
                         | <geometry type>
                         | <temporal type>
                         | <coverage type>
       <type operator> ::= {$typeOf: <name>}
                         | {$addAttrs: [<type>,<type>]}
                         | {$remAttrs: [<type>,<type>]}
                         | {$unset: <type>}
           <basic type> ::= unit|string|integer|real|boolean
      <geometry type> ::= point|linestring|polygon|geometrycollection|multipoint
                          multipoint|multilinestring|multipolygon|geometry|bbox
       <temporal type> ::= instant|period|multiinstant|regularmultiinstant|multiperiod
                          regularmultiperiod|duration
       <coverage type> ::= rectifiedgridcoverage|gridcoverage|referenceablegridcoverage|
                          multipointcoverage|multicurvecoverage|multisurfacecoverage
```

Table C.2: BNF for JSON serialization of pre- and postconditions

```
<expression> ::= {$and: [<expression>,...,<expression>]}
               | {$or: [<expression>,...,<expression>]}
               | {$not: <expression>}
               | {$implies: [<expression>,<expression>]}
               | {$<relation>:[<term>,<term>]}
               | {$<predicate>:[<term>,...,<term>]}
               | {$<quantifier>:[<variable>,<set>,<expression>]}
     <term> ::= {$<function>:[<term>,...,<term>]}
               | {$literal: <constant>}
               | <variable>
 <relation> ::= lt|lte|gt|gte|eq|df
<quantifier> ::= forall|exists
```

Appendix D

JSON Workflow Representation

Listing D.1 presents the RDF model of the workflow presented in Figure 6.1.

Listing D.1: RDF model for workflows

```
@prefix : <http://www.semanticweb.org/itc/ontologies/workflow#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://www.semanticweb.org/itc/ontologies/workflow> .

_:b0 rdf:type :wfs ;
  :hasName "Rivers" ;
  :hasUrl "http://localhost:3001/wfs/rivers" ;
  :hasOutput _:b1 .

_:b1 rdf:type :output ;
  :hasName "rivers" ;
  :sequenceFlow _:b6 .

_:b2 rdf:type :literal ;
  :hasValue 1000 ;
  :hasType "Real" ;
  :hasOutput _:b3 .

_:b3 rdf:type :output ,
  :hasName "literal" ;
  :sequenceFlow _:b7 .

_:b4 rdf:type :wps ;
  :hasName "Buffer" ;
  :hasUrl "http://localhost:3000/wps/buffer/job" ;
  :hasOutput _:b5 ;
  :hasInput _:b6 ,
            _:b7 ,
            _:b8 .

_:b5 rdf:type :output ;
  :hasName "buffered" ;
  :sequenceFlow _:b15 .

_:b6 rdf:type :input ;
  :hasName "features" .

_:b7 rdf:type :input ;
  :hasName "distance" .

_:b8 rdf:type :input ;
  :hasName "unit" .

_:b9 rdf:type :wfs ,
  :hasName "Buildings" ;
  :hasUrl "http://localhost:3001/wfs/buildings" ;
  :hasOutput _:b10 .

```

```
_:b10 rdf:type :output ,
  :hasName "buildings" ;
  :sequenceFlow _:b14 .

_:b11 rdf:type :wps ,
  :hasUrl "http://localhost:3000/wps/intersects/jobs" ;
  :hasName "Intersects" ;
  :hasOutput _:b12 ,
             _:b13
  :hasInput _:b14 ,
            _:b15 ;

_:b12 rdf:type :output ;
  :hasName "failed" .

_:b13 rdf:type :output ;
  :hasName "passed" .

_:b14 rdf:type :input ;
  :hasName "features" .

_:b15 rdf:type :input ;
  :hasName "filter" .
```

Listing D.2: Ontology for workflows

```
@prefix : <http://www.semanticweb.org/itc/ontologies/workflow#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://www.semanticweb.org/itc/ontologies/workflow> .

<http://www.semanticweb.org/itc/ontologies/workflow> rdf:type owl:Ontology .

:task rdf:type owl:Class .

:wfs rdf:type owl:Class ;
     rdfs:subClassOf :task .

:wps rdf:type owl:Class ;
     rdfs:subClassOf :task .

:literal rdf:type owl:Class ;
         rdfs:subClassOf :task .

:conditional rdf:type owl:Class ;
             rdfs:subClassOf :task .

:subgraph rdf:type owl:Class ;
          rdfs:subClassOf :task .

:iterateInput rdf:type owl:Class ;
              rdfs:subClassOf :task .

:iterateSets rdf:type owl:Class ;
             rdfs:subClassOf :task .

:iterateMultivalue rdf:type owl:Class ;
                   rdfs:subClassOf :task .

:inputParameter rdf:type owl:Class ;
                rdfs:subClassOf :task .

:outputParameter rdf:type owl:Class ;
                 rdfs:subClassOf :task .

```

```
:input rdf:type owl:Class .

:output rdf:type owl:Class .

:condition rdf:type owl:Class .

:metadata rdf:type owl:Class .

:hasInput rdf:type owl:ObjectProperty ;
          rdfs:range :input ;
          rdfs:domain :task .

:hasOutput rdf:type owl:ObjectProperty ;
           rdfs:range :output ;
           rdfs:domain :task .

:sequenceFlow rdf:type owl:ObjectProperty ;
              rdfs:range :input ;
              rdfs:domain :output .

:hasCondition rdf:type owl:ObjectProperty ;
              rdfs:domain :conditional ;
              rdfs:range :condition .

:hasMetadata rdf:type owl:ObjectProperty ;
             rdfs:domain :task ;
             rdfs:range :metadata .

:hasName rdf:type owl:DatatypeProperty ;
         rdfs:domain :task ;
         rdfs:range xsd:string .

:hasType rdf:type owl:DatatypeProperty ;
         rdfs:domain :literal ;
         rdfs:range xsd:string .

:hasReference rdf:type owl:DatatypeProperty ;
              rdfs:range xsd:string ;
              rdfs:domain [ rdf:type owl:Class ;
                            owl:unionOf ( :inputParameter
                                          :outputParameter
                                        )
                          ] .

:hasUrl rdf:type owl:DatatypeProperty ;
        rdfs:range xsd:string ;
        rdfs:domain [ rdf:type owl:Class ;
                      owl:unionOf ( :wfs
                                    :wps
                                    :subgraph
                                    :iterateInput
                                    :iterateSets
                                    :iterateMultivalue
                                  )
                    ] .

:hasValue rdf:type owl:DatatypeProperty ;
          rdfs:domain :literal .
```

Listing D.3: JSON-LD context for workflow

```
@context:{
  orchestration: "http://www.opengis.net/ont/jsonw/",
  tasks: {@id: "orchestration:tasks/", @container: "@set"},
  sequenceFlows: {@id: "orchestration:sequenceFlows/", @container: "@set"},
  id: "orchestration:id/",
  name: "orchestration:name/",
  inputs: {@id: "orchestration:inputs/", @container: "@set"},
```

```
  outputs: {@id: "orchestration:outputs/", @container: "@set"},
  url: "orchestration:url/",
  value: "orchestration:value/",
  valueType: "orchestration:valueType/",
        condition: "orchestration:condition/",
        reference: "orchestration:reference/",
        metadata: "orchestration:metadata/",
  from: "orchestration:from/",
  to: "orchestration:to/",
  fromPort: "orchestration:fromPort/",
  toPort: "orchestration:toPort/",
  wfs: "orchestration:wfs/",
  wps: "orchestration:wps/",
  conditional: "orchestration:conditional/",
  subgraph: "orchestration:subgraph/",
  iterateInput: "orchestration:iterateInput/",
  iterateSets: "orchestration:iterateSets/",
  iterateMultivalue: "orchestration:iterateMultivalue/",
  inputParameter: "orchestration:inputParameter/",
  outputParameter: "orchestration:outputParameter/",
  literal: "orchestration:literal/"
}
```